EXHIBIT 1

UNITED STATES DISTRICT COURT

for the

Western District of Texas

K.Mizra LLC *Plaintiff* v. Cisco Systems, Inc. *Defendant*	)	Civil Action No. 6:20-cv-1031-ADA

SUBPOENA TO TESTIFY AT A DEPOSITION IN A CIVIL ACTION

To: Microsoft Corporation c/o Corporation Service Company
300 Deschutes Way SW, Ste 208 MC-CSC1, Tumwater, WA 98501

(Name of person to whom this subpoena is directed)

☑ *Testimony:* **YOU ARE COMMANDED** to appear at the time, date, and place set forth below to testify at a deposition to be taken in this civil action. If you are an organization, you must designate one or more officers, directors, or managing agents, or designate other persons who consent to testify on your behalf about the following matters, or those set forth in an attachment:

See Attachment A.

Place: Veritext Seattle 1200 Fifth Avenue, Suite 1820 Seattle, WA 98101	Date and Time: 09/27/2021 9:00 am

The deposition will be recorded by this method: Videographic and/or stenographic

☑ *Production:* You, or your representatives, must also bring with you to the deposition the following documents, electronically stored information, or objects, and must permit inspection, copying, testing, or sampling of the material: See requests for documents listed in Attachment A.

The following provisions of Fed. R. Civ. P. 45 are attached – Rule 45(c), relating to the place of compliance; Rule 45(d), relating to your protection as a person subject to a subpoena; and Rule 45(e) and (g), relating to your duty to respond to this subpoena and the potential consequences of not doing so.

Date: 08/06/2021

CLERK OF COURT

OR

/s/ Cliff Win, Jr.

Signature of Clerk or Deputy Clerk — *Attorney's signature*

The name, address, e-mail address, and telephone number of the attorney representing *(name of party)* K.Mizra LLC, who issues or requests this subpoena, are:

Cliff Win, Folio Law Group, 14512 Edgewater Ln NE, lake Forest Park, WA 98155, cliff.win@foliolaw.com, 415-340-2035

Notice to the person who issues or requests this subpoena

If this subpoena commands the production of documents, electronically stored information, or tangible things before trial, a notice and a copy of the subpoena must be served on each party in this case before it is served on the person to whom it is directed. Fed. R. Civ. P. 45(a)(4).

Civil Action No. 6:20-cv-1031-ADA

PROOF OF SERVICE

(This section should not be filed with the court unless required by Fed. R. Civ. P. 45.)

I received this subpoena for *(name of individual and title, if any)* ______________________
on *(date)* ____________ .

❒ I served the subpoena by delivering a copy to the named individual as follows: ______________
____
______________________________________ on *(date)* ____________ ; or

❒ I returned the subpoena unexecuted because: ______________________________
____ .

Unless the subpoena was issued on behalf of the United States, or one of its officers or agents, I have also tendered to the witness the fees for one day's attendance, and the mileage allowed by law, in the amount of
$ ______________ .

My fees are $ ____________ for travel and $ ____________ for services, for a total of $ 0.00 .

I declare under penalty of perjury that this information is true.

Date: ____________

____
Server's signature

____
Printed name and title

____
Server's address

Additional information regarding attempted service, etc.:

Print Save As... Add Attachment Reset

Federal Rule of Civil Procedure 45 (c), (d), (e), and (g) (Effective 12/1/13)

(c) Place of Compliance.

(1) *For a Trial, Hearing, or Deposition.* A subpoena may command a person to attend a trial, hearing, or deposition only as follows:
(A) within 100 miles of where the person resides, is employed, or regularly transacts business in person; or
(B) within the state where the person resides, is employed, or regularly transacts business in person, if the person
(i) is a party or a party's officer; or
(ii) is commanded to attend a trial and would not incur substantial expense.

(2) *For Other Discovery.* A subpoena may command:
(A) production of documents, electronically stored information, or tangible things at a place within 100 miles of where the person resides, is employed, or regularly transacts business in person; and
(B) inspection of premises at the premises to be inspected.

(d) Protecting a Person Subject to a Subpoena; Enforcement.

(1) *Avoiding Undue Burden or Expense; Sanctions.* A party or attorney responsible for issuing and serving a subpoena must take reasonable steps to avoid imposing undue burden or expense on a person subject to the subpoena. The court for the district where compliance is required must enforce this duty and impose an appropriate sanction—which may include lost earnings and reasonable attorney's fees—on a party or attorney who fails to comply.

(2) *Command to Produce Materials or Permit Inspection.*
(A) *Appearance Not Required.* A person commanded to produce documents, electronically stored information, or tangible things, or to permit the inspection of premises, need not appear in person at the place of production or inspection unless also commanded to appear for a deposition, hearing, or trial.
(B) *Objections.* A person commanded to produce documents or tangible things or to permit inspection may serve on the party or attorney designated in the subpoena a written objection to inspecting, copying, testing, or sampling any or all of the materials or to inspecting the premises—or to producing electronically stored information in the form or forms requested. The objection must be served before the earlier of the time specified for compliance or 14 days after the subpoena is served. If an objection is made, the following rules apply:
(i) At any time, on notice to the commanded person, the serving party may move the court for the district where compliance is required for an order compelling production or inspection.
(ii) These acts may be required only as directed in the order, and the order must protect a person who is neither a party nor a party's officer from significant expense resulting from compliance.

(3) *Quashing or Modifying a Subpoena.*

(A) *When Required.* On timely motion, the court for the district where compliance is required must quash or modify a subpoena that:

(i) fails to allow a reasonable time to comply;
(ii) requires a person to comply beyond the geographical limits specified in Rule 45(c);
(iii) requires disclosure of privileged or other protected matter, if no exception or waiver applies; or
(iv) subjects a person to undue burden.
(B) *When Permitted.* To protect a person subject to or affected by a subpoena, the court for the district where compliance is required may, on motion, quash or modify the subpoena if it requires:
(i) disclosing a trade secret or other confidential research, development, or commercial information; or
(ii) disclosing an unretained expert's opinion or information that does not describe specific occurrences in dispute and results from the expert's study that was not requested by a party.
(C) *Specifying Conditions as an Alternative.* In the circumstances described in Rule 45(d)(3)(B), the court may, instead of quashing or modifying a subpoena, order appearance or production under specified conditions if the serving party:
(i) shows a substantial need for the testimony or material that cannot be otherwise met without undue hardship; and
(ii) ensures that the subpoenaed person will be reasonably compensated.

(e) Duties in Responding to a Subpoena.

(1) *Producing Documents or Electronically Stored Information.* These procedures apply to producing documents or electronically stored information:
(A) *Documents.* A person responding to a subpoena to produce documents must produce them as they are kept in the ordinary course of business or must organize and label them to correspond to the categories in the demand.
(B) *Form for Producing Electronically Stored Information Not Specified.* If a subpoena does not specify a form for producing electronically stored information, the person responding must produce it in a form or forms in which it is ordinarily maintained or in a reasonably usable form or forms.
(C) *Electronically Stored Information Produced in Only One Form.* The person responding need not produce the same electronically stored information in more than one form.
(D) *Inaccessible Electronically Stored Information.* The person responding need not provide discovery of electronically stored information from sources that the person identifies as not reasonably accessible because of undue burden or cost. On motion to compel discovery or for a protective order, the person responding must show that the information is not reasonably accessible because of undue burden or cost. If that showing is made, the court may nonetheless order discovery from such sources if the requesting party shows good cause, considering the limitations of Rule 26(b)(2)(C). The court may specify conditions for the discovery.

(2) *Claiming Privilege or Protection.*
(A) *Information Withheld.* A person withholding subpoenaed information under a claim that it is privileged or subject to protection as trial-preparation material must:
(i) expressly make the claim; and
(ii) describe the nature of the withheld documents, communications, or tangible things in a manner that, without revealing information itself privileged or protected, will enable the parties to assess the claim.
(B) *Information Produced.* If information produced in response to a subpoena is subject to a claim of privilege or of protection as trial-preparation material, the person making the claim may notify any party that received the information of the claim and the basis for it. After being notified, a party must promptly return, sequester, or destroy the specified information and any copies it has; must not use or disclose the information until the claim is resolved; must take reasonable steps to retrieve the information if the party disclosed it before being notified; and may promptly present the information under seal to the court for the district where compliance is required for a determination of the claim. The person who produced the information must preserve the information until the claim is resolved.

(g) Contempt.
The court for the district where compliance is required—and also, after a motion is transferred, the issuing court—may hold in contempt a person who, having been served, fails without adequate excuse to obey the subpoena or an order related to it.

For access to subpoena materials, see Fed. R. Civ. P. 45(a) Committee Note (2013).

ATTACHMENT A

Instructions

1. These requests are continuing in nature. You are requested to promptly supplement your responses as responsive documents or information is created or discovered or asresponsive documents or information come into your possession, custody, or control.

2. If there is no responsive, non-privileged document or information for any request, state so in your response.

3. If you object to any request on the basis that the request is unduly burdensome, specifically describe the undue burden.

4. If you find any request or definition ambiguous, identify the specific ambiguity and describe what construction you used in your response.

5. Provide a complete privilege log for any document that you assert is protected by the attorney-client privilege, work product immunity, or other protection.

6. When an asserted privilege does not extend to the whole document, produce the non-privileged part of the document. Clearly and prominently indicate the privileged portion of the document.

7. When a document responsive to the following requests contains (or has been revised to include) any postscript, notation, change, amendment, or addendum not appearing on the document itself as originally prepared, each revised original, draft, copy, or reproduction shall be produced as a different document.

8. When there is anything physically or electronically attached, affixed, appended, linked, or joined to a document, produce that thing as well.

9. To bring within the scope of these requests any and all conceivably relevant matters

or information which might otherwise be construed to be outside the scope:

a. the singular of each word shall be construed to include its plural and vice versa;

b. "and" and "or" shall be construed both conjunctively as well as disjunctively, and thus both shall be construed as "and/or";

c. "each" shall be construed to include "every" and vice versa;

d. "any" shall be construed to include "all" and vice versa;

e. the present tense shall be construed to include the past tense and vice versa;

f. the masculine shall be construed to include the feminine and vice versa; and

10. The documents produced in response to these requests shall be produced either organized and designated to correspond to categories in the requests or as they are maintained in the normal course of business. Further:

a. all associated file labels, file headings, and file folders shall be produced together with the responsive documents from each file;

b. all documents that cannot be legibly copied shall be produced in their original form, otherwise produce multipage TIFF images;

c. all multipage TIFF images shall be organized and grouped to indicate document boundaries as the originals; and

d. each page shall be given a discrete production number.

11. When asked to identify a person employed by you, state that person's name and title. When asked to identify a person not currently employed by you, state the person's name and information sufficient to locate and contact such person.

12. When asked to identify a document, state the production number associated with the document. If there is no production number, state the identity of the persons who prepared it, the nature and substance of the document with sufficient particularity to enable it to be identified, the date of the document, and the document's last known location and custodian.

13. When asked to "Explain," fully describe, including with reference to the underlying facts, persons, and documents (including the production numbers of any documents).

14. When asked to identify, describe, or explain on a "patent-by-patent" basis, that means separately provide the identification, description, or explanation for each such patent.

15. None of the instructions, definitions, or requests is an admission related to the existence, relevance, or admissibility of any evidence, or to the truth or accuracy of any statement or characterization.

16. To the extent a request involves a time period (e.g., documents created or dated prior to August 13, 2011) and you cannot determine the proper date for a document or when the document was created, include such documents.

Definitions

1. "You" means Microsoft Corporation and any subsidiaries, parents, divisions, predecessors, successors, and any and all affiliated entities of the foregoing.

2. "K.Mizra" means K.Mizra LLC and any subsidiaries, parents, divisions, predecessors, successors, affiliated entities, or related entities.

3. "Cisco" means Cisco Systems, Inc. and any subsidiaries, parents, divisions, predecessors, successors, affiliated entities, or related entities.

4. "Communications" and "communications" means all transmissions of information of any kind, orally, in writing, or in any other manner, at any time or place, and under any circumstances whatsoever.

5. "Concerning" and "concerning" means referring to, relating to, comprising, constituting, reflecting, mentioning, evidencing, representing, discussing, describing, pertaining to, or connected with the stated subject matter.

6. “Document” and “document” means anything within the scope of Federal Rule of Civil Procedure 34, including without limitation all types of documents, electronically stored information, and tangible things.

7. “Source Code” means a text listing in a form readable by a source code editor tool of computer instructions, commands, and data definitions as well as any configuration files expressed in a form suitable for input to an assembler, compiler, or other translator.

8. “Windows Operating Systems” means the term, “operating system,” including versions Windows Vista and later (e.g. Windows 7, 8, 10, and 11), as described in the following:

- https://download.microsoft.com/download/1/c/9/1c9813b8-089c-4fef-b2ad-ad80e79403ba/Whitepaper%20-%20The%20Windows%2010%20random%20number%20generation%20infrastructure.pdf [Exhibit A, “The Windows 10 random number generation infrastructure”]
- https://www.microsoft.com/security/blog/2019/11/25/going-in-depth-on-the-windows-10-random-number-generation-infrastructure/ [Exhibits B, “Going in-depth on the Windows 10 random number generation infrastructure”]
- https://opdhsblobprod04-secondary.blob.core.windows.net/contents/4960c3bad9654704b3e1888df466a99c/5e3c1ffce0fb12fa38fc285c7a23f45a?sv=2018-03-28&sr=b&si=ReadPolicy&sig=1g3RK%2FDvgh7Ux35HpqFoxYWC4pnKD%2F9SfA8iLW5nsGw%3D&st=2021-07-21T23%3A30%3A24Z&se=2021-07-22T23%3A40%3A24Z [Exhibit C, “Information protection”]
- https://download.microsoft.com/download/c/1/5/c150e1ca-4a55-4a7e-94c5-bfc8c2e785c5/Windows%2010%20Minimum%20Hardware%20Requirements.pdf [Exhibit D, “Windows 10 Minimum Hardware Requirements”] (excerpted)

9. “Trusted Platform Modules” and “TPM” mean the terms, “trusted platform module” or “TPM,” as described in Exhibits A, B, C, and D.

10. “BCrypt” means the API described in: https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/ [Exhibit E].

11. “NCrypt” means the API described in: https://docs.microsoft.com/en-

us/windows/win32/api/ncrypt/ [Exhibit F].

12. "WinCrypt" means the API described in: https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/ [Exhibit G].

13. "Crypt32" and "Advapi32" mean the APIs from the DLLs described in: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/ee391633(v=vs.85) [Exhibit H, "Server Core Functions by DLL"] (excerpted).

14. The "root PRNG," "SystemPrng," "ProcessPrng," "BCryptGenRandom," "CryptGenRandom," and "RtlGenRandom" mean the random number generation APIs described in Exhibit A, here in: https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3094.pdf [Exhibit I, "Microsoft Windows FIPS 140 Validation"], and here in: https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp1329.pdf [Exhibit J, "Microsoft Windows 7 Cryptographic Primitives Library (bcryptprimitives.dll) Security Policy Document"].

15. "Digital Certificates" means the terms, "certificate" and "keys," as described in Exhibit C, "Information protection" at "Trusted Platform Module Technology Overview," "TPM fundamentals," and "How Windows 10 uses the Trusted Platform Module," and in: https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/manage/component-updates/tpm-key-attestation [Exhibit K, "TPM Key Attestation"].

16. "Key Storage Provider" or "KSP" means code that enables key storage and retrieval, as described in Exhibit K.

17. "Platform Crypto Provider" means code that implements cryptographic standards and algorithms, as described in: https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptopenalgorithmprovider [Exhibit L], and in "Using the Windows 8 Platform Crypto

Provider and Associated TPM Functionality," available here, https://www.microsoft.com/en-us/download/details.aspx?id=52487 [Exhibit M] (excerpted).

18. "CNG" or "CNG.SYS" means the Cryptography API Next Generation as described in Exhibits A, I, and J.

19. "TPM Base Services" or "TBS" are the system services described in: https://docs.microsoft.com/en-us/windows/win32/tbs/about-tbs [Exhibit N].

20. "TBS Submit Command Function" means the function described in: https://docs.microsoft.com/en-us/windows/win32/api/tbs/nf-tbs-tbsip_submit_command [Exhibit O].

Requests for Documents

1. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing the cryptographic and security functions and structures on Windows Operating Systems, including for the following APIs: BCrypt, NCrypt, WinCrypt, Crypt32, and Advapi32 (collectively, the "Security APIs").

2. Documents and Source Code relating to or describing the operation of Windows random number generation functions that use the Trusted Platform Modules (TPM) on Windows Operating Systems (e.g., for example, by using random bytes from the TPM as a source of entropy for a seed, either directly or from an entropy pool), including the root PRNG, SystemPrng, ProcessPrng, BCryptGenRandom. CryptGenRandom, and RtlGenRandom (collectively, the "Secure RNGs").

3. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing Platform Crypto Providers on Windows Operating Systems, including: the BCrypt Platform Crypto Provider and the providers

enumerated as MS_PLATFORM_CRYPTO_PROVIDER and MS_PRIMITIVE_PROVIDER.

4. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in the storage of Digital Certificates and keys on Trusted Platform Modules (TPM) for Windows Operating Systems.

5. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing Key Storage Providers on Windows Operating Systems, including: the PCPStorageProvider.

6. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing source code that gathers random values from the TPM as an entropy source to seed/reseed any of the Secure RNGs, as described in Exhibit A at 9.

7. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing the source code in Winload that retrieves TPM randomness that is then hashed and used to seed an SP 800-90 AES-CTR-DRBG, which then produces output that is passed to CNG at startup, as described in Exhibit A at 11.

8. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing the source code in CNG that instantiates the root PRNG using the output passed by Winload, as described in Exhibit A at 11.

9. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing code that gathers random values from the TPM as an entropy source, including within Winload, CNG.SYS, and the TPM driver, as described in Exhibit A at 11-13 and Exhibit H at 18.

10. Documents and Source Code relating to or describing the operation of headers,

functions, libraries, and DLL files involved in implementing the source code located in CNG.SYS that gathers entropy from sources, including from random values gathered from the TPM, as described in Exhibit I at 18.

11. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing code that gathers random values from the TPM as an entropy source to store in entropy pools, as described in Exhibit A at 9.

12. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing code that gathers random values from the TPM as an entropy source to contribute to entropy pools by implementing functions such as, EntropyRegisterSource and EntropyProvideData, including from CNG.SYS and the TPM driver, as described in Exhibit A at 11-13 and Exhibit H at 18-19.

13. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing the source code located in CNG.SYS that provides RNG to bcryptprimitives.dll, as described in Exhibit I at 8 and Exhibit J at 8.

14. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files involved in implementing the source code located in bcryptprimitives.dll that receives RNG from CNG.SYS, as described in Exhibit I at 8 and Exhibit J at 8.

15. Documents and Source Code relating to or describing the operation of headers, functions, libraries, and DLL files that invoke TBS APIs, including the TBS Submit Command Function.

16. Documents sufficient to show the approximate date when the Security APIs (identified by specific function) first used a TPM.

17. Documents sufficient to show the approximate date when the Secure RNGs first used a TPM, for example, by using the random bytes from the TPM.

18. Documents sufficient to show the approximate date when You first became aware that customers were using a TPM.

19. Documents sufficient to show the approximate percentage of usage by version of Windows Operating Systems (e.g, Windows Vista, Windows 7, 8, 10, and 11) as of the date of first usage of the TPM (either by the Security APIs, Secure RNGs, or customers) until the present.

20. Documents sufficient to authenticate Exhibit A, "The Windows 10 random number generation infrastructure."

21. Documents sufficient to authenticate Exhibit B, "Going in-depth on the Windows 10 random number generation infrastructure."

22. Documents sufficient to authenticate Exhibit C, "Information Protection."

23. Documents sufficient to authenticate Exhibit D, "Windows 10 Minimum Hardware Requirements."

24. Documents sufficient to authenticate Exhibit E, "Bcrypt,h header."

25. Documents sufficient to authenticate Exhibit F, "Ncrypt.h header."

26. Documents sufficient to authenticate Exhibit G, "Wincrypt.h header."

27. Documents sufficient to authenticate Exhibit H, "Server Core Functions by DLL (Windows)."

28. Documents sufficient to authenticate Exhibit I, "Microsoft Windows FIPS 140 Validation."

29. Documents sufficient to authenticate Exhibit J, "Microsoft Windows 7 Cryptographic Primitives Library (bcryptprimitives.dll) Security Policy Document."

30. Documents sufficient to authenticate Exhibit K, "TPM Key Attestation,"

31. Documents sufficient to authenticate Exhibit L, "BCryptOpenAlgorithmProvider Function (bcrypt.h)."

32. Documents sufficient to authenticate Exhibit M, "Using the Windows 8 Platform

Crypto Provider and Associated TPM Functionality."

33. Documents sufficient to authenticate Exhibit N, "About TBS."

34. Documents sufficient to authenticate Exhibit O, "Tbsip_Submit_Command Function (tbs.h)."

Deposition Topics

1. Authentication of documents produced pursuant to Requests for Documents above.

2. Identification and operation of headers, functions, libraries, and DLL files involved in implementing the cryptographic and security functions and structures on Windows Operating Systems including for the following APIs: BCrypt, NCrypt, WinCrypt, Crypt32, and Advapi32 (collectively, the "Security APIs").

3. Identification and operation of Windows random number generation functions that use the Trusted Platform Module (TPM) on Windows Operating Systems (e.g., for example, by using random bytes from the TPM as a source of entropy for a seed, either directly or from an entropy pool), including the root PRNG, SystemPrng, ProcessPrng, BCryptGenRandom, CryptGenRandom, and RtlGenRandom (collectively, the "Secure RNGs").

4. Identification and operation of headers, functions, libraries, and DLL files involved in implementing Platform Crypto Providers on Windows Operating Systems, including: the BCrypt Platform Crypto Provider and the providers enumerated as MS_PLATFORM_CRYPTO_PROVIDER and MS_PRIMITIVE_PROVIDER.

5. Identification and operation of headers, functions, libraries, and DLL files involved in the storage of Digital Certificates and keys on Trusted Platform Modules (TPM) for Windows Operating Systems.

6. Identification and operation of headers, functions, libraries, and DLL files involved

in implementing Key Storage Providers on Windows Operating Systems, including: the PCPStorageProvider.

7. Identification and operation of headers, functions, libraries, and DLL files involved in implementing code that gathers random values from the TPM as an entropy source to seed/reseed any of the Secure RNGs, as described in Exhibit A at 9.

8. Identification and operation of headers, functions, libraries, and DLL files involved in implementing the source code in Winload that retrieves TPM randomness that is then hashed and used to seed an SP 800-90 AES-CTR-DRBG, which then produces output that is passed to CNG at startup, as described in Exhibit A at 11.

9. Identification and operation of headers, functions, libraries, and DLL files involved in implementing the source code in CNG that instantiates the root PRNG using the output passed by Winload, as described in Exhibit A at 11.

10. Identification and operation of headers, functions, libraries, and DLL files involved in implementing code that gathers random values from the TPM as an entropy source, including within Winload, CNG.SYS, and the TPM driver, as described in Exhibit A at 11-13 and Exhibit H at 18.

11. Identification and operation of headers, functions, libraries, and DLL files involved in implementing the source code located in CNG.SYS that gathers entropy from sources, including from random values gathered from the TPM, as described in Exhibit I at 18.

12. Identification and operation of headers, functions, libraries, and DLL files involved in implementing code that gathers random values from the TPM as an entropy source to store in entropy pools, as described in Exhibit A at 9.

13. Identification and operation of headers, functions, libraries, and DLL files involved

in implementing code that gathers random values from the TPM as an entropy source to contribute to entropy pools by implementing functions such as, EntropyRegisterSource and EntropyProvideData, including from CNG.SYS and the TPM driver, as described in Exhibit A at 11-13 and Exhibit H at 18-19.

14. Identification and operation of headers, functions, libraries, and DLL files involved in implementing the source code located in CNG.SYS that provides RNG to bcryptprimitives.dll, as described in Exhibit I at 8 and Exhibit J at 8.

15. Identification and operation of headers, functions, libraries, and DLL files involved in implementing the source code located in bcryptprimitives.dll that receives RNG from CNG.SYS, as described in Exhibit I at 8 and Exhibit J at 8.

16. Identification and operation of headers, functions, libraries, and DLL files that invoke TBS APIs, including the TBS Submit Command Function.

17. The approximate date when the Security APIs (identified by specific function) first used a TPM.

18. The approximate date when the Secure RNGs first used a TPM, for example, by using the random bytes from the TPM.

19. The approximate date when You first became aware that customers were using a TPM.

20. The approximate percentage of usage by version of Windows Operating Systems (e.g, Windows Vista, Windows 7, 8, 10, and 11) as of the date of first usage of the TPM (either by the Security APIs, Secure RNGs, or customers) until the present.

EXHIBIT A

Microsoft

The Windows 10 random number generation infrastructure

Niels Ferguson

Published: October 2019

For the latest information, please see
https://aka.ms/win10rng

This document describes the Windows 10 random number generation infrastructure. It is intended for readers who are familiar with random number generators and entropy collection terminology.

PRNG infrastructure

The PRNG infrastructure generates random numbers for all areas of the OS based on a single seed in the kernel. We will discuss the seeding of the PRNG infrastructure in the second part of the document.

Basic PRNG

All PRNGs in the system are SP800-90 AES_CTR_DRBG with 256-bit security strength using the df() function for seeding and re-seeding (see SP 800-90 for details).

AES_CTR_DRBG is a cryptographic pseudo-random number generator. That means that given any number of output bytes, it is computationally infeasible to determine the internal state of the PRNG or any other output byte. As the name suggests, AES_CTR_DRBG is based on using the AES block cipher in a counter mode, similarly to the way it is used to create a key stream in AES-GCM.

This PRNG has *backtracking resistance*; after it has produced an output, the updated PRNG state does not contain enough information to recover that output. The backtracking resistance property is maintained throughout the RNG system.

Buffered PRNG

The Basic PRNGs are not used directly, but rather through a wrapping layer that adds several features.

- A small buffer of random bytes to improve performance for small requests.
- A lock to support multi-threading.
- A seed version.

Buffering

The buffering is straightforward. There is a small buffer (currently 128 bytes). If a request for random bytes is 128 bytes or larger, it is generated directly from AES_CTR_DRGB. If it is smaller than 128 bytes it is taken from the buffer. The buffer is re-filled from the AES_CTR_DRBG whenever it runs empty. So, if the buffer contains 4 bytes and the request is for 8 bytes, the 4 bytes are taken from the buffer, the buffer is refilled with 128 bytes, and the first 4 bytes of the refilled buffer are used to complete the request, leaving 124 bytes in the buffer. When bytes are taken from the buffer, their locations are wiped (zeroed). This maintains the backtracking resistance. The bytes in the buffer are

discarded (i.e. not used but not wiped) on every reseed; the next time a small request arrives the buffer is overwritten with new random bytes.

(The implementation will also occasionally throw away bytes in the buffer to improve the memory alignment of requests.)

We give a short sketch of the security analysis of this buffer construction:

- Bytes in the buffer are from the DRBG. Bytes are used only once, and wiped when they are given out. Thus, every output byte is a byte from the DRBG; the buffering merely re-orders the bytes. The re-ordering is determined by the sequence of requests for random numbers. These requests can depend on previous random bytes, but not on the random bytes currently in the buffer. As we assume that all output bytes of the DRBG are independent and equivalent, the re-ordering does not change the independence or probability distribution of the output bytes.
- The buffer maintains secrecy. After a call to generate bytes, the buffered RNG state no longer has the data to reconstruct the output it provided. Of course, at any time the RNG state has the information to predict all future outputs until the next reseed.
- Discarded bytes (on reseed or to improve alignment) are not wiped. These bytes are never given as output to a caller. However, it would be perfectly okay for a caller to have requested those bytes, and published them. Thus, there is no security concern with not wiping discarded bytes.

Rationale: Each request from an AES_CTR_DRBG has a significant overhead as it requires an AES key expansion, producing the output bytes, and producing 48 additional bytes for the next DRBG state. For large requests, the overhead is not significant, but for small requests the overhead cost dominates. The buffer allows the system to amortize the overhead over multiple requests, significantly reducing the average cost for small requests.

Locking

All access to the buffered PRNG state is gated on the lock. This ensures that multiple threads do not attempt to read or modify the same state at the same time.

Root PRNG

There is a single (buffered) Root PRNG in kernel mode maintained by the CNG.SYS driver. All random bytes in the system are derived in some way from this PRNG. The Root PRNG is instantiated upon system startup. The entropy system (described below) initializes and reseeds the root PRNG.

Kernel per-processor PRNG

Using a single root PRNG would create a bottleneck on large computers. To avoid a bottleneck, the kernel mode has one (buffered) PRNG state per logical processor. (A 4-core CPU with hyperthreading has 8 logical processors from the OS point of view.) The per-processor PRNG states are also maintained by the CNG.SYS driver.

When a caller asks for random data in kernel mode, the code determines what logical CPU it is running on and checks if a PRNG state for this CPU has already been allocated. If it has, it uses that PRNG state to provide the requested random bytes. If there is no per-CPU state for the current CPU it attempts to allocate a PRNG state and instantiate it with seed material from the root PRNG. If that succeeds, it proceeds as before, servicing the request from the newly-created PRNG. If the allocation fails, the request is serviced from the root PRNG.

In practice, the allocations fail almost never and all requests are served by the per-CPU state. This has a number of very nice properties. The number of PRNG states scales with the size of the machine, providing both low memory footprint on small machines and high throughput on big machines. The memory for each per-processor PRNG state is allocated from that NUMA node which improves memory locality. Using a per-processor state also has good cache-locality, and there is very low contention on the PRNG state lock.

We also have the property that a request for random bytes *never fails*. In the past our RNG functions could return an error code. We have observed that there are many callers that never check for the error code, even if they are generating cryptographic key material. This can lead to serious security vulnerabilities if an attacker manages to create a situation in which the RNG infrastructure returns an error. For that reason, the Win10 RNG infrastructure will never return an error code and always produce high-quality random bytes for every request.

Process base PRNG

For each user-mode process, we have a (buffered) base PRNG maintained by BCryptPrimitives.dll. When this DLL loads it requests a random seed from kernel mode (where it is produced by the per-CPU states) and seeds the process base PRNG. If this were to fail, BCryptPrimitive.dll fails to load, which in most cases causes the process to terminate. This behavior ensures that we never have to return an error code from the RNG system.

Process per-processor PRNG

Just like in kernel mode, we have per-processor buffered PRNG states in each process. These are again created on-demand and seeded from the process base PRNG. The per-processor PRNG states are maintained by BCryptPrimitives.dll.

A running machine can have thousands of PRNG states. If the machine has N logical CPUs, and M processes, then there can be up to (N+1)(M+1) PRNG states. This could grow to a very large number on very large machines, but this has not been a problem in

practice. (Very large machines tend to have very large memories, and they tend to run a few very large processes rather than thousands of smaller ones.)

Reseeding the tree

All PRNG states in Windows 10 are reseeded on a regular schedule.

The root PRNG keeps a seed version. This is a 64-bit count of how many times the root PRNG has been seeded since boot. This counter value is published in a memory location that is readable and writable from kernel mode, and readable (but not writeable) in user mode.

All buffered PRNG states, both in user mode and kernel mode, keep track of their current seed version. A seed version of X implies that the PRNG was reseeded with data derived from the root PRNG when the root had seed version X. Any time a PRNG produces seed data for another PRNG, it also provides its current seed version, which the target PRNG stores.

Any time a PRNG is asked for random data, it checks its seed version against the root PRNG's seed version. (This check is very quick as it only involves a memory read and a 64-bit comparison.) If they are not equal, the PRNG will first reseed from its parent before producing output.

As an example, let's assume that we have a full set of PRNGs with seed version 7, and the root PRNG is just being reseeded by the entropy system. The root PRNG will increment its seed version to 8. A few microseconds later an application asks for 13 random bytes.

The application's thread selects the user-mode per-processor PRNG state and asks for 13 random bytes. This state checks its seed version and finds that it is out of date. It asks its parent, the process base PRNG for 32 random bytes to reseed with. The process base PRNG finds that it is out of date and performs an IOCTL request to the kernel for 32 random bytes to reseed itself. The IOCTL request asks the kernel per-processor PRNG state, which finds itself out of date and asks the root PRNG for seed material. The root is always up-to-date (by definition) so it provides reseed data to the kernel per-processor PRNG state. This reseeds, and provides the seed material to the process base PRNG, which reseeds and provides seed material to the user-mode per-processor PRNG state, which reseeds and provides the applications with the requested 13 bytes.

Subsequent requests require much less work. Once a PRNG has been reseeded it does not need to reseed until the next time the root PRNG reseeds. Furthermore, each PRNG in the system caches up to 128 bytes of output, so the first time a PRNG is asked for seed material it will generate 128 bytes and hand out 32 bytes. The next time it is asked for 32 bytes of seed material, it can be provided from the buffer. For example, if another thread on another CPU in the same process asks for random data, it will only reseed the per-processor state for that CPU. The seed material will be copied from the buffer of the process base CPU, making this a much faster operation.

This on-demand reseeding has the same effect as if we were to forcefully reseed every PRNG in the system whenever the root PRNG was reseeded, but it is far more efficient, and it avoids updating PRNG states that are not being used at all.

There is, however, a significant cost to reseeding the root PRNG. With hundreds of processes the system can contain thousands of PRNG states. Thus, the total cost of a reseed is quite high. Therefore, frequent root PRNG reseeding are avoided.

Random number generation APIs

The following are the primary APIs for random number generation in Windows 10.

SystemPrng

This is a function exported by CNG.SYS and available to all code in kernel mode. It is the primary interface to the kernel-mode per-processor PRNG system.

ProcessPrng

This is the primary interface to the user-mode per-processor PRNGs. It is implemented in BCryptPrimitives.dll.

BCryptGenRandom

The BCRYPT_RNG_ALGORITHM produces random data by calling the per-processor AES-CTR-DRBG PRNGs. This works both in kernel mode and user mode.

The BCRYPT_RNG_FIPS186_DSA_ALGORITHM and BCRYPT_RNG_DUAL_EC_ALGORITHM are no longer supported. For compatibility, these algorithms still work, but they produce output from the same per-processor AES-CTR-DRBG states as the default RNG algorithm.

CryptGenRandom

The Microsoft CSPs use ProcessPrng function to produce random bytes.

RtlGenRandom

RtlGenRandom uses the ProcessPrng function to produce random bytes.

Entropy system

The entropy system consists of several components:

- Entropy sources
- Entropy pools that store entropy from the entropy sources
- Reseed logic that determines when and how to reseed the root PRNG from the pools

Entropy sources

There is a kernel API for creating new entropy sources. Windows 10 supports three types:

- Low pull
- High pull
- High push

There are minor differences in how the entropy provided by different entropy source types is handled. In general, low sources are assumed to provide unconditioned entropy events; such as mouse movements. High sources are assumed to provide high-quality random bytes. All sources can provide entropy data at any time, but a pull source is additionally notified whenever the system reseeds the root PRNG from the entropy pools. This allows an on-demand source to provide entropy for every reseed without having its own timer logic.

For any source, the first time it provides entropy the data is used to directly reseed the root PRNG. The entropy pools are bypassed. The primary reason is to ensure that as soon as an entropy source provides data, PRNG output depends on the data from that source. Furthermore, it allows an external caller to force a reseed of the PRNG tree.

All subsequent times that a source provides entropy, the entropy is put in the entropy pools.

As mentioned above, reseeding all PRNGs in the system is expensive; we recommend against frequent creation of new entropy sources.

Reseeding the root

The root is reseeded from the pools on a time schedule. After the initial seeding at boot time (documented below) the first reseed is scheduled 1 second later. The interval between reseeds is tripled every time, so subsequent reseeds happen 3, 9, 27, etc. seconds after the previous one. The interval keeps increasing until it hits the limit, which is currently at 3600 seconds (1 hour). Detail: All timers used for this schedule have a 33% variability; this allows the OS to trigger the timer at a point in time when the CPU is already awake and avoids having to wake the CPU to only process a reseed. (Waking up the CPU is expensive in terms of power.)

Rationale: At early boot we want to get as much entropy into the system as soon as we can. On the other hand, reseeds are very expensive (both in time and power consumption), and if we are in a situation where the flow of entropy is low, we want to collect more entropy between reseeds. The progressive slow-down of the reseed schedule is a compromise between these goals.

Multiple entropy pools

Each entropy pool is implemented as a SHA-512 computation; all data provided to the entropy pool is appended, and when the pool is used the SHA-512 of all the data is used as output of the pool.

The system can have multiple entropy pools. When there are N pools, entropy events from each source are distributed round-robin fashion over the N different pools. This ensures that each pool has approximately the same inflow of entropy.

The N pools are numbered 0,…,N-1. Pool 0 is used on every reseed. Pool 1 is used (in addition to pool 0) every 3rd reseed, pool 2 is used every 9th reseed (in addition to pools 0 and 1), etc. In general, pool k is used every 3^kth reseed (in addition to all the lower numbered pools).

On startup, only one pool exists, and none of the multi-pool logic is enabled until the reseed interval hits the maximum. Once the reseed interval hits the maximum, the creation of additional entropy pools is enabled. A new pool is created whenever it would have been used in a reseed if it had existed. Thus, once the reseed interval hits the maximum, there are two more reseeds with just one pool. On the third reseed a second pool would have been used, so it is created. From that point forward entropy events are divided between the pools, and pool 1 is used every 3rd reseed. Similarly, 9 reseeds later the 3rd pool is enabled.

The design rationale of the multiple entropy pool system is given in an appendix.

Distributing entropy over the pools

By default each entropy source distributes its entropy events over the pools in a round-robin fashion. If there are 3 pools, consecutive entropy events are put in pools 0, 1, 2, 0, 1, 2, ….

The only difference between the handling of entropy from low and high entropy sources is that the first 32 bytes produced by a high entropy source after a reseed from the pools is always put in pool 0. Thus, a high entropy source will preferentially affect the next reseed.

A typical high-pull source will be active just after every reseed, and it will produce 64 bytes of high-quality random data. The first 32 bytes go into pool 0; the remaining bytes go into the next pool in the round-robin schema.

Initial seeding

Initial seeding starts in Winload before the Ntoskrnl has been loaded. Winload retrieves the following items

- Seed file
- External entropy
- TPM randomness
- RDRAND randomness
- ACPI-OEM0 table
- Output from the UEFI entropy provider
- The current time

The individual entropy sources are discussed later.

The data from these sources is hashed together (using SHA-512) and winload uses the result to seed a SP 800-90 AES-CTR-DRBG. This DRBG produces 48 bytes of output that is passed on to CNG to be used at startup. (Other output bytes are passed to the kernel for future use.) Winload also passes the per-source results to CNG to be included in diagnostic event logs.

When CNG loads, it instantiates the root PRNG using the 48 bytes passed by winload. It then starts each of its own entropy sources (described below). Any entropy source that provides data will of course trigger a root PRNG reseed. Note that the AES-CTR-DRBG reseed function combines the existing state of the PRNG with the input so that the existing state is not lost.

Entropy sources

Windows 10 has many entropy sources; these work together to ensure that the OS has good entropy. Different entropy sources guarantee good entropy in different situations; by using them all the best coverage is attained.

Interrupt timings

The primary entropy source in Windows 10 is the interrupt timings. On each interrupt to a CPU the interrupt hander gets the Time Stamp Count (TSC) from the CPU. This is typically a counter that runs on the CPU clock frequency; on X86 and X64 CPUs this is done using the RDTSC instruction.

Let TSC[i] be the TSC value for interrupt i on a 64-bit CPU. To reduce the amount of data, multiple TSC values are first combined using the following rule:

```
n := floor(i/64) mod 32
B[n] <- (B[n] >>> 19) xor TSC[i]
```

Here B is an array of 32 words of 64 bits each. An equivalent way to describe this is that 64 consecutive TSC values are each rotated by a different rotation amount, the results xorred together, and the result xorred into the B[n] word.

Every 1024 interrupts half of the B array has been updated and the interrupt handler schedules a DPC that calls back to the interrupt timing entropy source code. This code copies out the 128 bytes that were just updated and feeds them to the entropy system as an entropy event. While this is ongoing, the interrupt handler routine is updating the other half of the B array.

The total amortized cost of gathering the interrupt timings is around 30 cycles/interrupt; about 25 in the actual interrupt handler and about 5 in the further processing.

On a 32-bit CPUs the same system is used, but the B array consists of 64 words of 32 bits and the rotation constant is 5.

Startup

On startup the interrupt handlers start collecting data before the CNG driver is loaded. When the CNG driver registers its callback with the kernel, the kernel calls the callback once for every logical CPU. The CNG entropy source code combines all that information (containing the collected TSC data from all the interrupts that have occurred so far) and provides that as the first entropy event (which then bypasses the pool and is used to reseed the root). This ensures that as much entropy as is available is pushed into the root PRNG on startup.

On a typical boot cycle there are hundreds of interrupts available at startup.

Analysis

The interrupt handler code has a feature that allows a special driver to collect the raw TSC data. Essentially the driver can set up a second array in which the interrupt handler will store the raw TSC data (in addition to the normal rotate-and-xor buffer operation). The driver can gather the data from the buffer during the callback and write the raw TSC data to disk.

Having gathered the raw data, statistical analysis by us and some of our customers indicate that TSC values have more than 1 bit of entropy each. As a typical machine will see hundreds of interrupts per second, the total entropy flow is quite high.

TPM

During boot, Winload gets 40 random bytes from the TPM (if present) and uses them as one of the entropy sources. This use of the TPM can be disabled through a BCDEDIT setting, and is also disabled during a safe boot. (This allows the OS to boot even on a machine where the BIOS can’t find the TPM and hangs when it is accessed.)

Once the OS is booted, the TPM driver loads, registers as an entropy source, and provides 64 bytes of entropy for each reseed. Due to TPM limitations, the TPM entropy source provides entropy at most once every 40 minutes.

RDRAND/RDSEED

The Intel RDRAND instruction is an on-demand high quality source of random data.

If the RDRAND instruction is present, Winload gathers 256 bits of entropy from the RDRAND instruction. Similarly, our kernel-mode code creates a high-pull source that provides 512 bits of entropy from the RDRAND instruction for each reseed. (As a high source, the first 256 bits are always put in pool 0; providing 512 bits ensures that the other pools also get entropy from this source.)

Due to some unfortunate design decisions in the internal RDRAND logic, the RDRAND instruction only provides random numbers with a 128-bit security level. The Win10 code tries to work around this limitation by gathering a large amount of output from RDRAND which should trigger a reseed of the RDRAND-internal PRNG to get more entropy. Whilst this solves the problem in most cases, it is possible for another thread to gather similar outputs form RDRAND which means that a 256-bit security level cannot be guaranteed.

Based on our feedback about this problem, Intel implemented the RDSEED instruction that gives direct access to the internal entropy source. When the RDSEED instruction is present, it is used in preference to RDRAND instruction which avoids the problem and provides the full desired guarantees. For each reseed, we gather 128 output bytes from RDSEED, hash them with SHA-512 to produce 64 output bytes. As explained before, 32 of these go into pool 0 and the others into the 'next' pool for this entropy source.

Seed file

The seed file is a registry entry in the system hive. It is written by the OS for use during the next reboot. This is the primary source of entropy for the OS during boot, except in the first boot after installation.

To write a new value for the seed file, the system requests 64 bytes from the system PRNG (this is the kernel-mode per-processor PRNG) and writes it to the registry.

A new seed file is written at startup, and during shutdown of the operating system. Additionally, if the machine did not shut down cleanly during the previous boot cycle, a new seed file is written periodically. Periodic writes start about 4 minutes after a reboot; subsequent writes are progressively slowed down (by factors of 3) until they reach once every 8 hours. This schedule is modified by delaying each seed file write until the next time the pools reseed the root PRNG; this avoids running an extra timer and waking up the CPU from a low-power state needlessly. The seed file write schedule is based on a clock that stops when the system is sleeping or hibernating.

On shutdown, the system reseeds the root PRNG with all the entropy in all the pools and writes a new seedfile. The seedfile is read on startup by winload.

Rationale: If the machine is shut down cleanly, all the entropy collected during the boot cycle is used to generate the seed file, and the next boot cycle will have good entropy at startup. However, if a machine is always shut down badly (e.g. powered down without an OS shutdown) then it would gather very little entropy. This mechanism detects that and ensures that the next boot cycle will regularly save fresh entropy to the seed file.

As the seedfile is updated during the boot process, it will be different for the next boot cycle even if the other entropy collected during the boot cycle is lost in an unclean power down. Furthermore, entropy collected by the interrupt timings between the initialization of the kernel and the writing of the seed file during startup is always used for the next boot.

The reason to disable periodic seedfile writes during normal operations is to avoid periodic disk activity which has a high cost in battery-powered devices that are always on.

External entropy

External entropy is a registry entry in the system hive. It is typically written by the setup program. When setup runs (typically from a WIN-PE image from the DVD or network) the underlying OS gathers entropy as usual. Towards the end of the setup, the setup process takes entropy from the PRNG system of the OS it is running on and writes it to the external entropy registry key in the newly installed OS. Winload consumes this entropy and it affects all subsequent RNG operations. If an external entropy value is present, it is deleted during the boot process.

Other applications can use this registry key to inject new entropy into an OS before it is booted. For diagnostic purposes the OS keeps a counter (in the registry) of how often it has encountered external entropy. This provides an easy diagnostic signal to ensure that the external entropy has been properly processed.

The external entropy is read by winload, and deleted by cng.sys upon startup.

This source ensures that the OS has good entropy if it was installed from a DVD or the network.

ACPI-OEM0

The ACPI-OEM0 is an ACPI table with the name OEM0. The Hyper-V hypervisor will create this table with 64 bytes of random data. The random data is derived from the host RNG infrastructure, and a fresh value is passed every time a guest is powered up. OEMs could provide this table on physical machines too, but we are not aware of any OEM doing that.

The ACPI-OEM0 table is read by winload. It ensures that any VM guest on Hyper-V has good entropy at startup. (This is especially important as many VMs are often launched from the same disk image, so they all get the same seedfile data.)

We do not know whether other hypervisors provide this ACPI table.

Firmware data

When the RNG system is first initialized, the CNG driver queries a number of system firmware table providers. It enumerates all tables, hashes all the data together, and provides that as a one-shot entropy source to the RNG system.

Currently the 'ACPI', 'RSMB' and 'FIRM' firmware table providers are queried.

Rationale: There is no particular reason to assume that these tables contain good entropy, but this data is cheap to gather and might contain machine-specific information which at least ensures that two machines that boot from the same disk image will produce different random numbers.

UEFI protocol

On UEFI machines there is a defined protocol to ask for random data from the UEFI drivers. Most ARM designs have a dedicated entropy source on the chipset. OEMs can implement this UEFI protocol to provide entropy to Windows at startup. We also recommend that they also expose the entropy source as a high-pull source to the entropy system to reseed the OS while it is running.

Time at startup

Winload gets the current time during every boot, and uses it as an entropy input.

Rationale: there is no reason to believe this provides any good entropy, but it greatly reduces the chance of two machines producing the same RNG state if they boot from the same disk image.

Virtual machine rewind

If the OS is running in a VM, there is a problem that most hypervisors can snapshot the state of the machine and later rewind the VM state to the saved state. This results in the machine running a second time with the exact same RNG state, which leads to serious security problems.

To reduce the window of vulnerability, Windows 10 on a Hyper-V VM will detect when the VM state is reset, retrieve a unique (not random) value from the hypervisor, and reseed the root RNG with that unique value. This does not eliminate the vulnerability, but it greatly reduces the time during which the RNG system will produce the same outputs as it did during a previous instantiation of the same VM state.

We do not know whether other hypervisors support this feature.

Appendix: Design rationale for multiple entropy pools

Of all the design aspects, the use of multiple pools has created the most discussion. This appendix explains the design rationale for using multiple pools.

Multiple entropy pools allow the system to reseed securely even if the rate of entropy is very low. The purpose of reseeds is to inject fresh entropy into the system so that an attacker that happens to know the current PRNG state will not know the future PRNG state. To achieve this goal, a reseed needs at least S bits of entropy, where S depends on the requirements but is typically in the range 128-256. (We'll use 128 in our examples.)

The traditional approach to ensuring a secure reseed is to run entropy estimation on the entropy provided by the sources. The system then collects entropy in a pool until it has at least S bits of entropy, and then uses that to reseed the PRNG system.

This approach is sound for dedicated entropy sources where the source can be characterized and analyzed. However, it fails for ad-hoc entropy sources such as those used by Windows. For example, the behavior of interrupt timings can be analyzed on existing hardware, but future hardware might have completely different properties so any entropy estimator tested on current hardware might be completely wrong on future hardware. (This is especially relevant if you consider that some of our customers consider the designers and implementers of future hardware as their opponents.) The danger is that a single bad entropy estimator might result in a system that consistently reseeds with, say, S/2 bits of true entropy and never achieves a secure state.

For this reason, Windows no longer uses entropy estimates to drive the reseed schedule. Rather, reseeds are done on a clock schedule; initially reseeds are done quickly to get entropy into the system quickly, but the reseeds are slowed down progressively to allow larger amounts of entropy to be collected between reseeds. The multi-pool system merely extends this behavior whilst at the same time retaining a minimum reseed speed.

For example: let's assume we have a machine whose entropy sources produce 30 bits of true entropy per hour. With one pool and one reseed each hour, each reseed gets 30 bits of entropy which is not a secure reseed. After 3 hours the second pool is enabled. Each pool now gets 15 bits of entropy per hour, so the next two reseeds (which use only pool 0) have 15 bits of entropy each. The reseed after that uses pools 0 and 1; pool 1 has collected 45 bits of entropy (3 hours of 15 bits each) and pool 0 has 15 bits, so the reseed has 60 bits of entropy. Essentially, the second pool has siphoned off some of the entropy from two reseeds and bunched it together into the third. Of course, 60 bits is not quite a secure reseed, so this pattern repeats three times after which the 3rd pool is enabled. Now each pool gets 10 bits per hour, and the reseeds will have in turn 10, 10, 40, 10, 10, 40, 10, 10, 130 bits of entropy. Finally, after about 21 hours the system has reseeded with 130 bits of entropy and is in a secure state.

It would of course be more efficient to collect exactly 128 bits of entropy (in just over 4 hours) and use that to reseed. We can look at the efficiency of a reseed system by looking at how much actual entropy it needs to achieve a secure reseed.

A reseed schedule based on entropy estimation assumes that each source has an entropy estimate that *never* overestimates the entropy. As mentioned above, this is infeasible for ad-hoc entropy sources. In practice, implementations like Win7 use entropy estimators that deliberately underestimate the amount of entropy in an attempt to reduce (but not eliminate) the risk of overestimating. The Win7 entropy estimator for the entropy scavenger is based on statistical analysis of the scavenging results in the most pessimistic scenario, and then reports an estimate that is 5 times lower. On the one hand, there is no guarantee that this estimator will never overestimate the entropy. On the other hand, the resulting reseed schedule is inefficient. Even in the most pessimistic scenario the system uses 5 times more entropy than necessary for the reseed, and in all other scenarios (when more entropy is available) the system is even less efficient. Compare this to the multi-pool example above. It is about 5 times slower than an ideal schedule, so in the worst-case scenario it is as good as the Win7 scavenger entropy estimation. In all other scenarios, it is actually much better.

We can show that the multi-pool system is within a constant factor of optimal under a very wide range of scenarios. Windows has a maximum of 8 entropy pools. Let's assume we get E bits of entropy per hour, and all 8 pools are in use. (Pool 7 will be used in a reseed once every 2187 hours, or about once every 3 month.) Each pool receives E/8 bits per hour. The worst-case inefficiency (in this rather simple model) is computed in the table below:

Time between reseeds	Entropy used in reseed	Entropy needed by MP	Inefficiency
1	1/8 * E	8S	8
3	4/8 * E	¾ * 8S	¾ * 8
9	13/8 * E	9/13 * 8S	9/13 * 8
27	40/8 * E	27/40 * 8S	27/40 * 8

The first column shows the time T between two reseeds that are secure. The second column shows how much entropy the multi-pool system uses in that reseed, and the third column how much entropy it needs to collect in that time to ensure that the reseed is secure. The final column shows the inefficiency of the multi-pool system (the amount of entropy it needs compared to the ideal schedule if the entropy estimators were exactly right.) As shown, the inefficiency is almost constant, and in fact it is bounded. (With fewer pools, the system is more efficient.) Thus, the multiple pools provide a simple guarantee: assuming the flow of entropy is uniform and distributed over the pools, the system is guaranteed to reseed securely after having received no more than

K*S bits of entropy, for a modest constant K. (With the obvious limits that the secure reseed is never faster than 1 hour and can't be slower than 2187 hours.)

This guarantee holds irrespective of any entropy estimators. If multiple attackers are attacking the same system at the same time, then the efficiency guarantee holds with respect to all attackers simultaneously even though the actual entropy of each entropy events can be a completely different value for each attacker.

In short, the multi-pool construction provides simple security guarantees, is within a constant factor of optimal, and dispenses with the vulnerable entropy estimators whose validity cannot be guaranteed.

Of course, the advantage of multi-pool is only apparent when the rate of entropy is extraordinary low. This is by itself an undesirable scenario, but not one that can be ignored for an OS used in as many scenarios as Windows.

Though the description of the multi-pool system is rather complicated, the implementation is actually rather simple, adding fewer than 200 lines of code to the system.

EXHIBIT B

November 25, 2019

Going in-depth on the Windows 10 random number generation infrastructure

Enterprise & Security Team

Share

Throughout the years, we've had ongoing conversations with researchers, developers, and customers around our implementation of certain security features within the Windows operating system. Most recently, we have open-sourced our cryptography libraries as a way to contribute and show our continued support to the security community

For our most recent contribution, we have decided to go in-depth on our implementation of pseudo-random number generation in Windows 10.

We are happy to release to the public **The Windows 10 random number generation infrastructure white paper**.

This whitepaper explores details about the Windows 10 pseudo-random number generator (PRNG) infrastructure, and lists the primary RNG APIs. The whitepaper also explains how the entropy system works, what the entropy sources are, and how initial seeding works.

We expect academic and security researchers, as well as operating system developers and people with an in-depth understanding of random number generation, to get the most value out of this whitepaper. Note: Some of the terminology used in this whitepaper assumes prior knowledge of random number generators and entropy collection terms.

We welcome and look forward to your feedback on this whitepaper and the technologies it describes in the comments below. We also appreciate any reports of security vulnerabilities that you may find in our implementation.

Filed under:

Cybersecurity

You may also like these articles

July 15, 2021

Protecting customers from a private-sector offensive actor using 0-day exploits and DevilsTongue malware

The Microsoft Threat Intelligence Center (MSTIC) alongside the Microsoft Security Response Center (MSRC) has uncovered a private-sector offensive actor, or PSOA, that we are calling SOURGUM in possession of now-patched, Windows 0-day exploits (CVE-2021-31979 and CVE-2021-33771).

Read more ›

July 14, 2021

Microsoft delivers comprehensive solution to battle rise in consent phishing emails

Microsoft threat analysts are tracking a continued increase in consent phishing emails, also called illicit consent grants, that abuse OAuth request links in an attempt to trick recipients into granting attacker-owned apps permissions to access sensitive data.

Read more ›

July 14, 2021

MISA expands portfolio and looks ahead during Microsoft Inspire

MISA extends product portfolio, adds sessions for Microsoft Inspire, and more.

Read more ›

Get started with Microsoft Security

Microsoft is a leader in cybersecurity, and we embrace our responsibility to make the world a safer place.

LEARN MORE >

Get all the news, updates, and more at @MSFTSecurity

What's new

Surface Laptop 4

Surface Laptop Go

Surface Go 2

Surface Pro X

Surface Duo

Microsoft 365

Windows 10 apps

HoloLens 2

Microsoft Store

Account profile

Download Center

Microsoft Store support

Returns

Order tracking

Virtual workshops and training

Microsoft Store Promise

Financing

Education

Microsoft in education

Office for students

Office 365 for schools

Deals for students & parents

Microsoft Azure in education

Enterprise

Azure

AppSource

Automotive

Government

Healthcare

Manufacturing

Financial services

Retail

Developer

Microsoft Visual Studio

Windows Dev Center

Developer Center

Microsoft developer program

Channel 9

Microsoft 365 Dev Center

Microsoft 365 Developer Program

Microsoft Garage

Company

Careers

About Microsoft

Company news

Privacy at Microsoft

Investors

Diversity and inclusion

Accessibility

Security

English (United States)

Sitemap Contact Microsoft Privacy Terms of use Trademarks Safety & eco About our ads

EXHIBIT C

Contents

Information protection

3/5/2021 • 2 minutes to read • Edit Online

Learn more about how to secure documents and other data across your organization.

SECTION	DESCRIPTION
BitLocker	Provides information about BitLocker, which is a data protection feature that integrates with the operating system and addresses the threats of data theft or exposure from lost, stolen, or inappropriately decommissioned computers.
Encrypted Hard Drive	Encrypted Hard Drive uses the rapid encryption that is provided by BitLocker Drive Encryption to enhance data security and management.
Kernel DMA Protection	Kernel DMA Protection protects PCs against drive-by Direct Memory Access (DMA) attacks using PCI hot plug devices connected to PCI accessible ports, such as Thunderbolt™ 3 ports.
Protect your enterprise data using Windows Information Protection (WIP)	Provides info about how to create a Windows Information Protection policy that can help protect against potential corporate data leakage.
Secure the Windows 10 boot process	Windows 10 supports features to help prevent rootkits and bootkits from loading during the startup process.
Trusted Platform Module	Trusted Platform Module (TPM) technology is designed to provide hardware-based, security-related functions. A TPM chip is a secure crypto-processor that helps you with actions such as generating, storing, and limiting the use of cryptographic keys.

Trusted Platform Module

3/5/2021 • 2 minutes to read • Edit Online

Applies to

- Windows 10
- Windows Server 2016

Trusted Platform Module (TPM) technology is designed to provide hardware-based, security-related functions. A TPM chip is a secure crypto-processor that helps you with actions such as generating, storing, and limiting the use of cryptographic keys. The following topics provide details.

TOPIC	DESCRIPTION
Trusted Platform Module Overview	Provides an overview of the Trusted Platform Module (TPM) and how Windows uses it for access control and authentication.
TPM fundamentals	Provides background about how a TPM can work with cryptographic keys. Also describes technologies that work with the TPM, such as TPM-based virtual smart cards.
TPM Group Policy settings	Describes TPM services that can be controlled centrally by using Group Policy settings.
Back up the TPM recovery information to AD DS	For Windows 10, version 1511 and Windows 10, version 1507 only, describes how to back up a computer's TPM information to Active Directory Domain Services.
Troubleshoot the TPM	Describes actions you can take through the TPM snap-in, TPM.msc: view TPM status, troubleshoot TPM initialization, and clear keys from the TPM. Also, for TPM 1.2 and Windows 10, version 1507 or 1511, describes how to turn the TPM on or off.
Understanding PCR banks on TPM 2.0 devices	Provides background about what happens when you switch PCR banks on TPM 2.0 devices.
TPM recommendations	Discusses aspects of TPMs such as the difference between TPM 1.2 and 2.0, and the Windows 10 features for which a TPM is required or recommended.

Trusted Platform Module Technology Overview

7/19/2021 • 4 minutes to read • Edit Online

Applies to

- Windows 10
- Windows Server 2016
- Windows Server 2019

This topic for the IT professional describes the Trusted Platform Module (TPM) and how Windows uses it for access control and authentication.

Feature description

Trusted Platform Module (TPM) technology is designed to provide hardware-based, security-related functions. A TPM chip is a secure crypto-processor that is designed to carry out cryptographic operations. The chip includes multiple physical security mechanisms to make it tamper resistant, and malicious software is unable to tamper with the security functions of the TPM. Some of the key advantages of using TPM technology are that you can:

- Generate, store, and limit the use of cryptographic keys.
- Use TPM technology for platform device authentication by using the TPM's unique RSA key, which is burned into itself.
- Help ensure platform integrity by taking and storing security measurements.

The most common TPM functions are used for system integrity measurements and for key creation and use. During the boot process of a system, the boot code that is loaded (including firmware and the operating system components) can be measured and recorded in the TPM. The integrity measurements can be used as evidence for how a system started and to make sure that a TPM-based key was used only when the correct software was used to boot the system.

TPM-based keys can be configured in a variety of ways. One option is to make a TPM-based key unavailable outside the TPM. This is good to mitigate phishing attacks because it prevents the key from being copied and used without the TPM. TPM-based keys can also be configured to require an authorization value to use them. If too many incorrect authorization guesses occur, the TPM will activate its dictionary attack logic and prevent further authorization value guesses.

Different versions of the TPM are defined in specifications by the Trusted Computing Group (TCG). For more information, consult the TCG Web site.

Automatic initialization of the TPM with Windows 10

Starting with Windows 10, the operating system automatically initializes and takes ownership of the TPM. This means that in most cases, we recommend that you avoid configuring the TPM through the TPM management console, **TPM.msc**. There are a few exceptions, mostly related to resetting or performing a clean installation on a PC. For more information, see Clear all the keys from the TPM. We're no longer actively developing the TPM management console beginning with Windows Server 2019 and Windows 10, version 1809.

In certain specific enterprise scenarios limited to Windows 10, versions 1507 and 1511, Group Policy might be used to back up the TPM owner authorization value in Active Directory. Because the TPM state persists across operating system installations, this TPM information is stored in a location in Active Directory that is separate from computer objects.

Practical applications

Certificates can be installed or created on computers that are using the TPM. After a computer is provisioned, the RSA private key for a certificate is bound to the TPM and cannot be exported. The TPM can also be used as a replacement for smart cards, which reduces the costs associated with creating and disbursing smart cards.

Automated provisioning in the TPM reduces the cost of TPM deployment in an enterprise. New APIs for TPM management can determine if TPM provisioning actions require physical presence of a service technician to approve TPM state change requests during the boot process.

Antimalware software can use the boot measurements of the operating system start state to prove the integrity of a computer running Windows 10 or Windows Server 2016. These measurements include the launch of Hyper-V to test that datacenters using virtualization are not running untrusted hypervisors. With BitLocker Network Unlock, IT administrators can push an update without concerns that a computer is waiting for PIN entry.

The TPM has several Group Policy settings that might be useful in certain enterprise scenarios. For more info, see TPM Group Policy Settings.

New and changed functionality

For more info on new and changed functionality for Trusted Platform Module in Windows 10, see What's new in Trusted Platform Module?.

Device health attestation

Device health attestation enables enterprises to establish trust based on hardware and software components of a managed device. With device heath attestation, you can configure an MDM server to query a health attestation service that will allow or deny a managed device access to a secure resource.

Some things that you can check on the device are:

- Is Data Execution Prevention supported and enabled?
- Is BitLocker Drive Encryption supported and enabled?
- Is SecureBoot supported and enabled?

> **NOTE**
> Windows 10, Windows Server 2016 and Windows Server 2019 support Device Health Attestation with TPM 2.0. Support for TPM 1.2 was added beginning with Windows version 1607 (RS1). TPM 2.0 requires UEFI firmware. A computer with legacy BIOS and TPM 2.0 won't work as expected.

Supported versions for device health attestation

TPM VERSION	WINDOWS 10	WINDOWS SERVER 2016	WINDOWS SERVER 2019
TPM 1.2	>= ver 1607	>= ver 1607	Yes
TPM 2.0	Yes	Yes	Yes

Related topics

- Trusted Platform Module (list of topics)

- Details on the TPM standard (has links to features using TPM)
- TPM Base Services Portal
- TPM Base Services API
- TPM Cmdlets in Windows PowerShell
- Prepare your organization for BitLocker: Planning and Policies - TPM configurations
- Azure device provisioning: Identity attestation with TPM
- Azure device provisioning: A manufacturing timeline for TPM devices
- Windows 10: Enabling vTPM (Virtual TPM)
- How to Multiboot with Bitlocker, TPM, and a Non-Windows OS

TPM fundamentals

3/26/2021 • 11 minutes to read • Edit Online

Applies to

- Windows 10
- Windows Server 2016

This topic for the IT professional provides a description of the components of the Trusted Platform Module (TPM 1.2 and TPM 2.0) and explains how they are used to mitigate dictionary attacks.

A Trusted Platform Module (TPM) is a microchip designed to provide basic security-related functions, primarily involving encryption keys. The TPM is usually installed on the motherboard of a computer, and it communicates with the remainder of the system by using a hardware bus.

Computers that incorporate a TPM can create cryptographic keys and encrypt them so that they can only be decrypted by the TPM. This process, often called wrapping or binding a key, can help protect the key from disclosure. Each TPM has a master wrapping key, called the storage root key, which is stored within the TPM itself. The private portion of a storage root key or endorsement key that is created in a TPM is never exposed to any other component, software, process, or user.

You can specify whether encryption keys that are created by the TPM can be migrated or not. If you specify that they can be migrated, the public and private portions of the key can be exposed to other components, software, processes, or users. If you specify that encryption keys cannot be migrated, the private portion of the key is never exposed outside the TPM.

Computers that incorporate a TPM can also create a key that has not only been wrapped, but is also tied to certain platform measurements. This type of key can be unwrapped only when those platform measurements have the same values that they had when the key was created. This process is referred to as "sealing the key to the TPM." Decrypting the key is called unsealing. The TPM can also seal and unseal data that is generated outside the TPM. With this sealed key and software, such as BitLocker Drive Encryption, you can lock data until specific hardware or software conditions are met.

With a TPM, private portions of key pairs are kept separate from the memory that is controlled by the operating system. Keys can be sealed to the TPM, and certain assurances about the state of a system (assurances that define the trustworthiness of a system) can be made before the keys are unsealed and released for use. Because the TPM uses its own internal firmware and logic circuits to process instructions, it does not rely on the operating system, and it is not exposed to vulnerabilities that might exist in the operating system or application software.

For info about which versions of Windows support which versions of the TPM, see Trusted Platform Module technology overview. The features that are available in the versions are defined in specifications by the Trusted Computing Group (TCG). For more info, see the Trusted Platform Module page on the Trusted Computing Group website: Trusted Platform Module.

The following sections provide an overview of the technologies that support the TPM:

- Measured Boot with support for attestation
- TPM-based Virtual Smart Card
- TPM-based certificate storage
- TPM Cmdlets

- Physical presence interface
- TPM 1.2 states and initialization
- Endorsement keys
- TPM Key Attestation
- Anti-hammering

The following topic describes the TPM Services that can be controlled centrally by using Group Policy settings: TPM Group Policy Settings.

Measured Boot with support for attestation

The Measured Boot feature provides antimalware software with a trusted (resistant to spoofing and tampering) log of all boot components. Antimalware software can use the log to determine whether components that ran before it are trustworthy versus infected with malware. It can also send the Measured Boot logs to a remote server for evaluation. The remote server can initiate remediation actions by interacting with software on the client or through out-of-band mechanisms, as appropriate.

TPM-based Virtual Smart Card

The Virtual Smart Card emulates the functionality of traditional smart cards, but Virtual Smart Cards use the TPM chip that is available on an organization's computers, rather than requiring the use of a separate physical smart card and reader. This greatly reduces the management and deployment cost of smart cards in an enterprise. To the end user, the Virtual Smart Card is always available on the computer. If a user needs to use more than one computer, a Virtual Smart Card must be issued to the user for each computer. A computer that is shared among multiple users can host multiple Virtual Smart Cards, one for each user.

TPM-based certificate storage

The TPM can be used to protect certificates and RSA keys. The TPM key storage provider (KSP) provides easy, convenient use of the TPM as a way of strongly protecting private keys. The TPM KSP can be used to generate keys when an organization enrolls for certificates, and the KSP is managed by templates in the UI. The TPM can also be used to protect certificates that are imported from an outside source. TPM-based certificates can be used exactly as standard certificates with the added functionality that the certificate can never leave the TPM from which the keys were generated. The TPM can now be used for crypto-operations through Cryptography API: Next Generation (CNG). For more info, see Cryptography API: Next Generation.

TPM Cmdlets

You can manage the TPM using Windows PowerShell. For details, see TPM Cmdlets in Windows PowerShell.

Physical presence interface

For TPM 1.2, the TCG specifications for TPMs require physical presence (typically, pressing a key) for turning the TPM on, turning it off, or clearing it. These actions typically cannot be automated with scripts or other automation tools unless the individual OEM supplies them.

TPM 1.2 states and initialization

For TPM 1.2, there are multiple possible states. Windows 10 automatically initializes the TPM, which brings it to an enabled, activated, and owned state.

Endorsement keys

For a TPM to be usable by a trusted application, it must contain an endorsement key, which is an RSA key pair. The private half of the key pair is held inside the TPM, and it is never revealed or accessible outside the TPM.

Key attestation

TPM key attestation allows a certification authority to verify that a private key is actually protected by a TPM and that the TPM is one that the certification authority trusts. Endorsement keys which have been proven valid can be used to bind the user identity to a device. Moreover, the user certificate with a TPM attested key provides higher security assurance backed up by the non-exportability, anti-hammering, and isolation of keys provided by a TPM.

Anti-hammering

When a TPM processes a command, it does so in a protected environment, for example, a dedicated microcontroller on a discrete chip or a special hardware-protected mode on the main CPU. A TPM can be used to create a cryptographic key that is not disclosed outside the TPM, but is able to be used in the TPM after the correct authorization value is provided.

TPMs have anti-hammering protection that is designed to prevent brute force attacks, or more complex dictionary attacks, that attempt to determine authorization values for using a key. The basic approach is for the TPM to allow only a limited number of authorization failures before it prevents more attempts to use keys and locks. Providing a failure count for individual keys is not technically practical, so TPMs have a global lockout when too many authorization failures occur.

Because many entities can use the TPM, a single authorization success cannot reset the TPM's anti-hammering protection. This prevents an attacker from creating a key with a known authorization value and then using it to reset the TPM's protection. Generally, TPMs are designed to forget about authorization failures after a period of time so the TPM does not enter a lockout state unnecessarily. A TPM owner password can be used to reset the TPM's lockout logic.

TPM 2.0 anti-hammering

TPM 2.0 has well defined anti-hammering behavior. This is in contrast to TPM 1.2 for which the anti-hammering protection was implemented by the manufacturer, and the logic varied widely throughout the industry.

For systems with TPM 2.0, the TPM is configured by Windows to lock after 32 authorization failures and to forget one authorization failure every two hours. This means that a user could quickly attempt to use a key with the wrong authorization value 32 times. For each of the 32 attempts, the TPM records if the authorization value was correct or not. This inadvertently causes the TPM to enter a locked state after 32 failed attempts.

Attempts to use a key with an authorization value for the next two hours would not return success or failure; instead the response indicates that the TPM is locked. After two hours, one authorization failure is forgotten and the number of authorization failures remembered by the TPM drops to 31, so the TPM leaves the locked state and returns to normal operation. With the correct authorization value, keys could be used normally if no authorization failures occur during the next two hours. If a period of 64 hours elapses with no authorization failures, the TPM does not remember any authorization failures, and 32 failed attempts could occur again.

Windows 8 Certification does not require TPM 2.0 systems to forget about authorization failures when the system is fully powered off or when the system has hibernated. Windows does require that authorization failures are forgotten when the system is running normally, in a sleep mode, or in low power states other than off. If a Windows system with TPM 2.0 is locked, the TPM leaves lockout mode if the system is left on for two hours.

The anti-hammering protection for TPM 2.0 can be fully reset immediately by sending a reset lockout command

to the TPM and providing the TPM owner password. By default, Windows automatically provisions TPM 2.0 and stores the TPM owner password for use by system administrators.

In some enterprise situations, the TPM owner authorization value is configured to be stored centrally in Active Directory, and it is not stored on the local system. An administrator can launch the TPM MMC and choose to reset the TPM lockout time. If the TPM owner password is stored locally, it is used to reset the lockout time. If the TPM owner password is not available on the local system, the administrator needs to provide it. If an administrator attempts to reset the TPM lockout state with the wrong TPM owner password, the TPM does not allow another attempt to reset the lockout state for 24 hours.

TPM 2.0 allows some keys to be created without an authorization value associated with them. These keys can be used when the TPM is locked. For example, BitLocker with a default TPM-only configuration is able to use a key in the TPM to start Windows, even when the TPM is locked.

Rationale behind the defaults

Originally, BitLocker allowed from 4 to 20 characters for a PIN. Windows Hello has its own PIN for logon, which can be 4 to 127 characters. Both BitLocker and Windows Hello use the TPM to prevent PIN brute-force attacks.

The TPM can be configured to use Dictionary Attack Prevention parameters (lockout threshold and lockout duration) to control how many failed authorizations attempts are allowed before the TPM is locked out, and how much time must elapse before another attempt can be made.

The Dictionary Attack Prevention Parameters provide a way to balance security needs with usability. For example, when BitLocker is used with a TPM + PIN configuration, the number of PIN guesses is limited over time. A TPM 2.0 in this example could be configured to allow only 32 PIN guesses immediately, and then only one more guess every two hours. This totals a maximum of about 4415 guesses per year. If the PIN is 4 digits, all 9999 possible PIN combinations could be attempted in a little over two years.

Increasing the PIN length requires a greater number of guesses for an attacker. In that case, the lockout duration between each guess can be shortened to allow legitimate users to retry a failed attempt sooner, while maintaining a similar level of protection.

Beginning with Windows 10, version 1703, the minimum length for the BitLocker PIN was increased to 6 characters to better align with other Windows features that leverage TPM 2.0, including Windows Hello. To help organizations with the transition, beginning with Windows 10, version 1709 and Windows 10, version 1703 with the October 2017 cumulative update installed, the BitLocker PIN length is 6 characters by default, but it can be reduced to 4 characters. If the minimum PIN length is reduced from the default of six characters, then the TPM 2.0 lockout period will be extended.

TPM-based smart cards

The Windows TPM-based smart card, which is a virtual smart card, can be configured to allow sign in to the system. In contrast with physical smart cards, the sign-in process uses a TPM-based key with an authorization value. The following list shows the advantages of virtual smart cards:

- Physical smart cards can enforce lockout for only the physical smart card PIN, and they can reset the lockout after the correct PIN is entered. With a virtual smart card, the TPM's anti-hammering protection is not reset after a successful authentication. The allowed number of authorization failures before the TPM enters lockout includes many factors.
- Hardware manufacturers and software developers have the option to use the security features of the TPM to meet their requirements.
- The intent of selecting 32 failures as the lock-out threshold is so users rarely lock the TPM (even when learning to type new passwords or if they frequently lock and unlock their computers). If users lock the TPM, they must to wait two hours or use some other credential to sign in, such as a user name and password.

Related topics

- Trusted Platform Module (list of topics)
- TPM Cmdlets in Windows PowerShell
- TPM WMI providers
- Prepare your organization for BitLocker: Planning and Policies - TPM configurations

How Windows 10 uses the Trusted Platform Module

3/5/2021 • 23 minutes to read • Edit Online

The Windows 10 operating system improves most existing security features in the operating system and adds groundbreaking new security features such as Device Guard and Windows Hello for Business. It places hardware-based security deeper inside the operating system than previous Windows versions had done, maximizing platform security while increasing usability. To achieve many of these security enhancements, Windows 10 makes extensive use of the Trusted Platform Module (TPM). This article offers a brief overview of the TPM, describes how it works, and discusses the benefits that TPM brings to Windows 10—as well as the cumulative security impact of running Windows 10 on a PC that contains a TPM.

See also:

- Windows 10 Specifications
- TPM Fundamentals
- TPM Recommendations

TPM Overview

The TPM is a cryptographic module that enhances computer security and privacy. Protecting data through encryption and decryption, protecting authentication credentials, and proving which software is running on a system are basic functionalities associated with computer security. The TPM helps with all these scenarios and more.

Historically, TPMs have been discrete chips soldered to a computer's motherboard. Such implementations allow the computer's original equipment manufacturer (OEM) to evaluate and certify the TPM separate from the rest of the system. Although discrete TPM implementations are still common, they can be problematic for integrated devices that are small or have low power consumption. Some newer TPM implementations integrate TPM functionality into the same chipset as other platform components while still providing logical separation similar to discrete TPM chips.

TPMs are passive: they receive commands and return responses. To realize the full benefit of a TPM, the OEM must carefully integrate system hardware and firmware with the TPM to send it commands and react to its responses. TPMs were originally designed to provide security and privacy benefits to a platform's owner and users, but newer versions can provide security and privacy benefits to the system hardware itself. Before it can be used for advanced scenarios, a TPM must be provisioned. Windows 10 automatically provisions a TPM, but if the user reinstalls the operating system, he or she may need to tell the operating system to explicitly provision the TPM again before it can use all the TPM's features.

The Trusted Computing Group (TCG) is the nonprofit organization that publishes and maintains the TPM specification. The TCG exists to develop, define, and promote vendor-neutral, global industry standards that support a hardware-based root of trust for interoperable trusted computing platforms. The TCG also publishes the TPM specification as the international standard ISO/IEC 11889, using the Publicly Available Specification Submission Process that the Joint Technical Committee 1 defines between the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC).

OEMs implement the TPM as a component in a trusted computing platform, such as a PC, tablet, or phone. Trusted computing platforms use the TPM to support privacy and security scenarios that software alone cannot achieve. For example, software alone cannot reliably report whether malware is present during the system startup process. The close integration between TPM and platform increases the transparency of the startup

process and supports evaluating device health by enabling reliable measuring and reporting of the software that starts the device. Implementation of a TPM as part of a trusted computing platform provides a hardware root of trust—that is, it behaves in a trusted way. For example, if a key stored in a TPM has properties that disallow exporting the key, that key *truly cannot leave the TPM*.

The TCG designed the TPM as a low-cost, mass-market security solution that addresses the requirements of different customer segments. There are variations in the security properties of different TPM implementations just as there are variations in customer and regulatory requirements for different sectors. In public-sector procurement, for example, some governments have clearly defined security requirements for TPMs, whereas others do not.

Certification programs for TPMs—and technology in general—continue to evolve as the speed of innovation increases. Although having a TPM is clearly better than not having a TPM, Microsoft's best advice is to determine your organization's security needs and research any regulatory requirements associated with procurement for your industry. The result is a balance between scenarios used, assurance level, cost, convenience, and availability.

TPM in Windows 10

The security features of Windows 10 combined with the benefits of a TPM offer practical security and privacy benefits. The following sections start with major TPM-related security features in Windows 10 and go on to describe how key technologies use the TPM to enable or increase security.

Platform Crypto Provider

Windows includes a cryptography framework called *Cryptographic API: Next Generation* (CNG), the basic approach of which is to implement cryptographic algorithms in different ways but with a common application programming interface (API). Applications that use cryptography can use the common API without knowing the details of how an algorithm is implemented much less the algorithm itself.

Although CNG sounds like a mundane starting point, it illustrates some of the advantages that a TPM provides. Underneath the CNG interface, Windows or third parties supply a cryptographic provider (that is, an implementation of an algorithm) implemented as software libraries alone or in a combination of software and available system hardware or third-party hardware. If implemented through hardware, the cryptographic provider communicates with the hardware behind the software interface of CNG.

The Platform Crypto Provider, introduced in the Windows 8 operating system, exposes the following special TPM properties, which software-only CNG providers cannot offer or cannot offer as effectively:

• **Key protection**. The Platform Crypto Provider can create keys in the TPM with restrictions on their use. The operating system can load and use the keys in the TPM without copying the keys to system memory, where they are vulnerable to malware. The Platform Crypto Provider can also configure keys that a TPM protects so that they are not removable. If a TPM creates a key, the key is unique and resides only in that TPM. If the TPM imports a key, the Platform Crypto Provider can use the key in that TPM, but that TPM is not a source for making additional copies of the key or enabling the use of copies elsewhere. In sharp contrast, software solutions that protect keys from copying are subject to reverse-engineering attacks, in which someone figures out how the solution stores keys or makes copies of keys while they are in memory during use.

• **Dictionary attack protection**. Keys that a TPM protects can require an authorization value such as a PIN. With dictionary attack protection, the TPM can prevent attacks that attempt a large number of guesses to determine the PIN. After too many guesses, the TPM simply returns an error saying no more guesses are allowed for a period of time. Software solutions might provide similar features, but they cannot provide the same level of protection, especially if the system restarts, the system clock changes, or files on the hard disk that count failed guesses are rolled back. In addition, with dictionary attack protection, authorization values such as PINs can be shorter and easier to remember while still providing the same level of protection as more complex values when using software solutions.

These TPM features give Platform Crypto Provider distinct advantages over software-based solutions. A practical way to see these benefits in action is when using certificates on a Windows 10 device. On platforms that include a TPM, Windows can use the Platform Crypto Provider to provide certificate storage. Certificate templates can specify that a TPM use the Platform Crypto Provider to protect the key associated with a certificate. In mixed environments, where some computers might not have a TPM, the certificate template could simply prefer the Platform Crypto Provider over the standard Windows software provider. If a certificate is configured as not able to be exported, the private key for the certificate is restricted and cannot be exported from the TPM. If the certificate requires a PIN, the PIN gains the TPM's dictionary attack protection automatically.

Virtual Smart Card

Smart cards are highly secure physical devices that typically store a single certificate and the corresponding private key. Users insert a smart card into a built-in or USB card reader and enter a PIN to unlock it. Windows can then access the card's certificate and use the private key for authentication or to unlock BitLocker protected data volumes. Smart cards are popular because they provide two-factor authentication that requires both something the user has (that is, the smart card) and something the user knows (such as the smart card PIN). Smart cards are difficult to use, however, because they require purchase and deployment of both smart cards and smart card readers.

In Windows, the Virtual Smart Card feature allows the TPM to mimic a permanently inserted smart card. The TPM becomes "something the user has" but still requires a PIN. Although physical smart cards limit the number of PIN attempts before locking the card and requiring a reset, a virtual smart card relies on the TPM's dictionary attack protection to prevent too many PIN guesses.

For TPM-based virtual smart cards, the TPM protects the use and storage of the certificate private key so that it cannot be copied when it is in use or stored and used elsewhere. Using a component that is part of the system rather than a separate physical smart card can reduce total cost of ownership because it eliminates "lost card" and "card left at home" scenarios while still delivering the benefits of smart card–based multifactor authentication. For users, virtual smart cards are simple to use, requiring only a PIN to unlock. Virtual smart cards support the same scenarios that physical smart cards support, including signing in to Windows or authenticating for resource access.

Windows Hello for Business

Windows Hello for Business provides authentication methods intended to replace passwords, which can be difficult to remember and easily compromised. In addition, user name - password solutions for authentication often reuse the same user name – password combinations on multiple devices and services; if those credentials are compromised, they are compromised in many places. Windows Hello for Business provisions devices one by one and combines the information provisioned on each device (i.e., the cryptographic key) with additional information to authenticate users. On a system that has a TPM, the TPM can protect the key. If a system does not have a TPM, software-based techniques protect the key. The additional information the user supplies can be a PIN value or, if the system has the necessary hardware, biometric information, such as fingerprint or facial recognition. To protect privacy, the biometric information is used only on the provisioned device to access the provisioned key: it is not shared across devices.

The adoption of new authentication technology requires that identity providers and organizations deploy and use that technology. Windows Hello for Business lets users authenticate with their existing Microsoft account, an Active Directory account, a Microsoft Azure Active Directory account, or even non-Microsoft Identity Provider Services or Relying Party Services that support Fast ID Online V2.0 authentication.

Identity providers have flexibility in how they provision credentials on client devices. For example, an organization might provision only those devices that have a TPM so that the organization knows that a TPM protects the credentials. The ability to distinguish a TPM from malware acting like a TPM requires the following TPM capabilities (see Figure 1):

• **Endorsement key**. The TPM manufacturer can create a special key in the TPM called an *endorsement key*. An endorsement key certificate, signed by the manufacturer, says that the endorsement key is present in a TPM that the manufacturer made. Solutions can use the certificate with the TPM containing the endorsement key to confirm a scenario really involves a TPM from a specific TPM manufacturer (instead of malware acting like a TPM.

• **Attestation identity key**. To protect privacy, most TPM scenarios do not directly use an actual endorsement key. Instead, they use attestation identity keys, and an identity certificate authority (CA) uses the endorsement key and its certificate to prove that one or more attestation identity keys actually exist in a real TPM. The identity CA issues attestation identity key certificates. More than one identity CA will generally see the same endorsement key certificate that can uniquely identify the TPM, but any number of attestation identity key certificates can be created to limit the information shared in other scenarios.

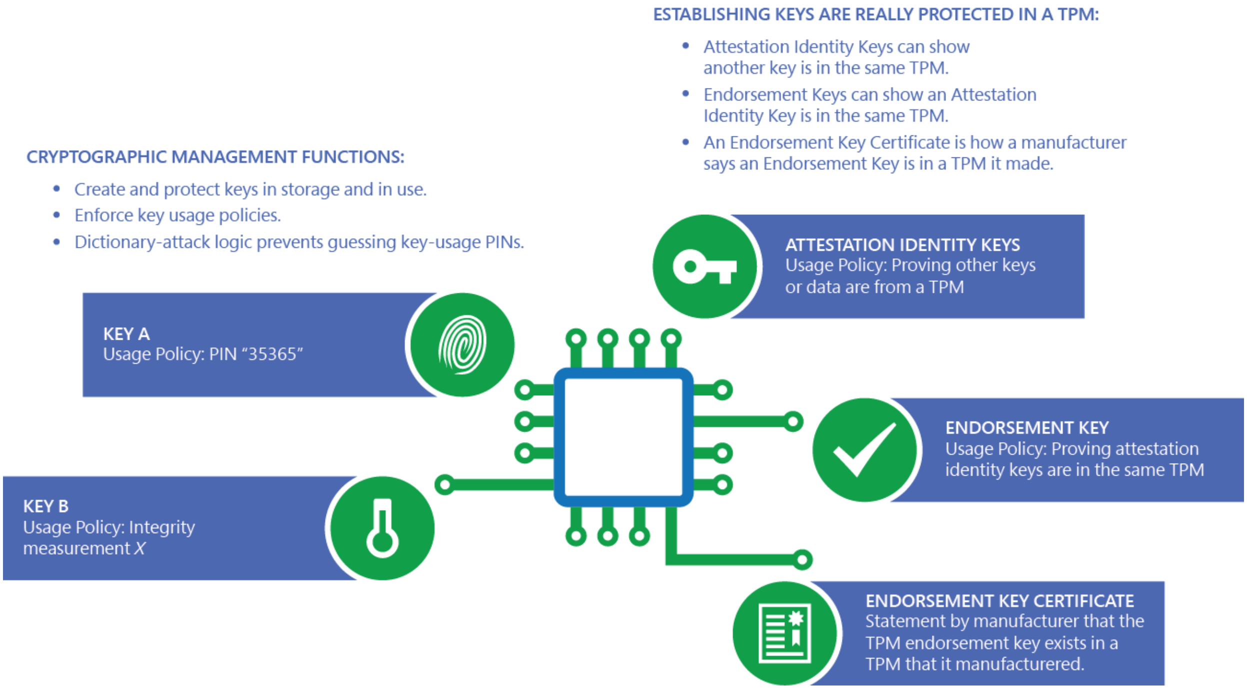


Figure 1: TPM Cryptographic Key Management

For Windows Hello for Business, Microsoft can fill the role of the identity CA. Microsoft services can issue an attestation identity key certificate for each device, user, and identify provider to ensure that privacy is protected and to help identity providers ensure that device TPM requirements are met before Windows Hello for Business credentials are provisioned.

BitLocker Drive Encryption

BitLocker provides full-volume encryption to protect data at rest. The most common device configuration splits the hard drive into several volumes. The operating system and user data reside on one volume that holds confidential information, and other volumes hold public information such as boot components, system information and recovery tools. (These other volumes are used infrequently enough that they do not need to be visible to users.) Without additional protections in place, if the volume containing the operating system and user data is not encrypted, someone can boot another operating system and easily bypass the intended operating system's enforcement of file permissions to read any user data.

In the most common configuration, BitLocker encrypts the operating system volume so that if the computer or hard disk is lost or stolen when powered off, the data on the volume remains confidential. When the computer is turned on, starts normally, and proceeds to the Windows logon prompt, the only path forward is for the user to log on with his or her credentials, allowing the operating system to enforce its normal file permissions. If something about the boot process changes, however—for example, a different operating system is booted from a USB device—the operating system volume and user data cannot be read and are not accessible. The TPM and

system firmware collaborate to record measurements of how the system started, including loaded software and configuration details such as whether boot occurred from the hard drive or a USB device. BitLocker relies on the TPM to allow the use of a key only when startup occurs in an expected way. The system firmware and TPM are carefully designed to work together to provide the following capabilities:

• **Hardware root of trust for measurement**. A TPM allows software to send it commands that record measurements of software or configuration information. This information can be calculated using a hash algorithm that essentially transforms a lot of data into a small, statistically unique hash value. The system firmware has a component called the Core Root of Trust for Measurement (CRTM) that is implicitly trusted. The CRTM unconditionally hashes the next software component and records the measurement value by sending a command to the TPM. Successive components, whether system firmware or operating system loaders, continue the process by measuring any software components they load before running them. Because each component's measurement is sent to the TPM before it runs, a component cannot erase its measurement from the TPM. (However, measurements are erased when the system is restarted.) The result is that at each step of the system startup process, the TPM holds measurements of boot software and configuration information. Any changes in boot software or configuration yield different TPM measurements at that step and later steps. Because the system firmware unconditionally starts the measurement chain, it provides a hardware-based root of trust for the TPM measurements. At some point in the startup process, the value of recording all loaded software and configuration information diminishes and the chain of measurements stops. The TPM allows for the creation of keys that can be used only when the platform configuration registers that hold the measurements have specific values.

• **Key used only when boot measurements are accurate**. BitLocker creates a key in the TPM that can be used only when the boot measurements match an expected value. The expected value is calculated for the step in the startup process when Windows Boot Manager runs from the operating system volume on the system hard drive. Windows Boot Manager, which is stored unencrypted on the boot volume, needs to use the TPM key so that it can decrypt data read into memory from the operating system volume and startup can proceed using the encrypted operating system volume. If a different operating system is booted or the configuration is changed, the measurement values in the TPM will be different, the TPM will not let Windows Boot Manager use the key, and the startup process cannot proceed normally because the data on the operating system cannot be decrypted. If someone tries to boot the system with a different operating system or a different device, the software or configuration measurements in the TPM will be wrong and the TPM will not allow use of the key needed to decrypt the operating system volume. As a failsafe, if measurement values change unexpectedly, the user can always use the BitLocker recovery key to access volume data. Organizations can configure BitLocker to store the recovery key in Active Directory Domain Services (AD DS).

Device hardware characteristics are important to BitLocker and its ability to protect data. One consideration is whether the device provides attack vectors when the system is at the logon screen. For example, if the Windows device has a port that allows direct memory access so that someone can plug in hardware and read memory, an attacker can read the operating system volume's decryption key from memory while at the Windows logon screen. To mitigate this risk, organizations can configure BitLocker so that the TPM key requires both the correct software measurements and an authorization value. The system startup process stops at Windows Boot Manager, and the user is prompted to enter the authorization value for the TPM key or insert a USB device with the value. This process stops BitLocker from automatically loading the key into memory where it might be vulnerable, but has a less desirable user experience.

Newer hardware and Windows 10 work better together to disable direct memory access through ports and reduce attack vectors. The result is that organizations can deploy more systems without requiring users to enter additional authorization information during the startup process. The right hardware allows BitLocker to be used with the "TPM-only" configuration giving users a single sign-on experience without having to enter a PIN or USB key during boot.

Device Encryption

Device Encryption is the consumer version of BitLocker, and it uses the same underlying technology. How it works is if a customer logs on with a Microsoft account and the system meets Modern Standby hardware requirements, BitLocker Drive Encryption is enabled automatically in Windows 10. The recovery key is backed up in the Microsoft cloud and is accessible to the consumer through his or her Microsoft account. The Modern Standby hardware requirements inform Windows 10 that the hardware is appropriate for deploying Device Encryption and allows use of the "TPM-only" configuration for a simple consumer experience. In addition, Modern Standby hardware is designed to reduce the likelihood that measurement values change and prompt the customer for the recovery key.

For software measurements, Device Encryption relies on measurements of the authority providing software components (based on code signing from manufacturers such as OEMs or Microsoft) instead of the precise hashes of the software components themselves. This permits servicing of components without changing the resulting measurement values. For configuration measurements, the values used are based on the boot security policy instead of the numerous other configuration settings recorded during startup. These values also change less frequently. The result is that Device Encryption is enabled on appropriate hardware in a user-friendly way while also protecting data.

Measured Boot

Windows 8 introduced Measured Boot as a way for the operating system to record the chain of measurements of software components and configuration information in the TPM through the initialization of the Windows operating system. In previous Windows versions, the measurement chain stopped at the Windows Boot Manager component itself, and the measurements in the TPM were not helpful for understanding the starting state of Windows.

The Windows boot process happens in stages and often involves third-party drivers to communicate with vendor-specific hardware or implement antimalware solutions. For software, Measured Boot records measurements of the Windows kernel, Early-Launch Anti-Malware drivers, and boot drivers in the TPM. For configuration settings, Measured Boot records security-relevant information such as signature data that antimalware drivers use and configuration data about Windows security features (e.g., whether BitLocker is on or off).

Measured Boot ensures that TPM measurements fully reflect the starting state of Windows software and configuration settings. If security settings and other protections are set up correctly, they can be trusted to maintain the security of the running operating system thereafter. Other scenarios can use the operating system's starting state to determine whether the running operating system should be trusted.

TPM measurements are designed to avoid recording any privacy-sensitive information as a measurement. As an additional privacy protection, Measured Boot stops the measurement chain at the initial starting state of Windows. Therefore, the set of measurements does not include details about which applications are in use or how Windows is being used. Measurement information can be shared with external entities to show that the device is enforcing adequate security policies and did not start with malware.

The TPM provides the following way for scenarios to use the measurements recorded in the TPM during boot:

• **Remote Attestation**. Using an attestation identity key, the TPM can generate and cryptographically sign a statement (or *quote*) of the current measurements in the TPM. Windows 10 can create unique attestation identity keys for various scenarios to prevent separate evaluators from collaborating to track the same device. Additional information in the quote is cryptographically scrambled to limit information sharing and better protect privacy. By sending the quote to a remote entity, a device can attest which software and configuration settings were used to boot the device and initialize the operating system. An attestation identity key certificate can provide further assurance that the quote is coming from a real TPM. Remote attestation is the process of recording measurements in the TPM, generating a quote, and sending the quote information to another system that evaluates the measurements to establish trust in a device. Figure 2 illustrates this process.

When new security features are added to Windows, Measured Boot adds security-relevant configuration information to the measurements recorded in the TPM. Measured Boot enables remote attestation scenarios that reflect the system firmware and the Windows initialization state.

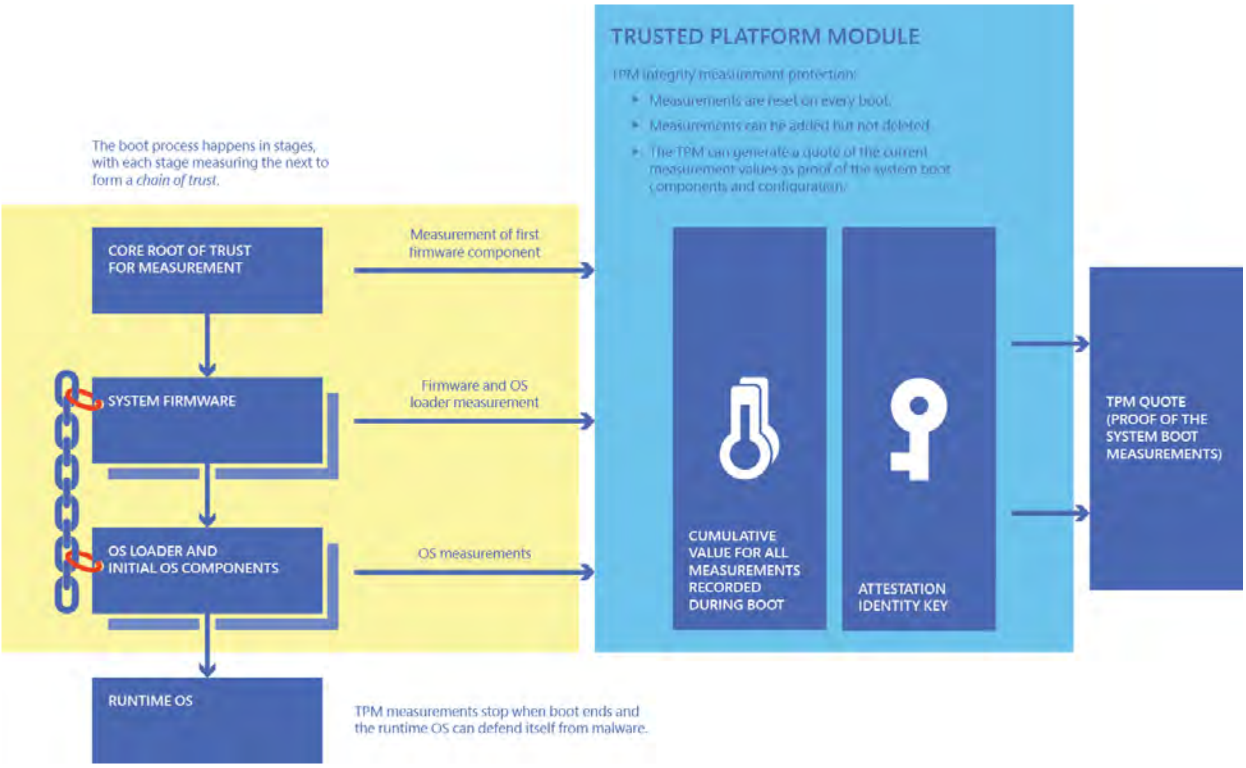


Figure 2: Process used to create evidence of boot software and configuration using a TPM

Health Attestation

Some Windows 10 improvements help security solutions implement remote attestation scenarios. Microsoft provides a Health Attestation service, which can create attestation identity key certificates for TPMs from different manufacturers as well as parse measured boot information to extract simple security assertions, such as whether BitLocker is on or off. The simple security assertions can be used to evaluate device health.

Mobile device management (MDM) solutions can receive simple security assertions from the Microsoft Health Attestation service for a client without having to deal with the complexity of the quote or the detailed TPM measurements. MDM solutions can act on the security information by quarantining unhealthy devices or blocking access to cloud services such as Microsoft Office 365.

Credential Guard

Credential Guard is a new feature in Windows 10 that helps protect Windows credentials in organizations that have deployed AD DS. Historically, a user's credentials (e.g., logon password) were hashed to generate an authorization token. The user employed the token to access resources that he or she was permitted to use. One weakness of the token model is that malware that had access to the operating system kernel could look through the computer's memory and harvest all the access tokens currently in use. The attacker could then use harvested tokens to log on to other machines and collect more credentials. This kind of attack is called a "pass the hash" attack, a malware technique that infects one machine to infect many machines across an organization.

Similar to the way Microsoft Hyper-V keeps virtual machines (VMs) separate from one another, Credential Guard uses virtualization to isolate the process that hashes credentials in a memory area that the operating system kernel cannot access. This isolated memory area is initialized and protected during the boot process so that components in the larger operating system environment cannot tamper with it. Credential Guard uses the TPM to protect its keys with TPM measurements, so they are accessible only during the boot process step when the separate region is initialized; they are not available for the normal operating system kernel. The local security

authority code in the Windows kernel interacts with the isolated memory area by passing in credentials and receiving single-use authorization tokens in return.

The resulting solution provides defense in depth, because even if malware runs in the operating system kernel, it cannot access the secrets inside the isolated memory area that actually generates authorization tokens. The solution does not solve the problem of key loggers because the passwords such loggers capture actually pass through the normal Windows kernel, but when combined with other solutions, such as smart cards for authentication, Credential Guard greatly enhances the protection of credentials in Windows 10.

Conclusion

The TPM adds hardware-based security benefits to Windows 10. When installed on hardware that includes a TPM, Window 10 delivers remarkably improved security benefits. The following table summarizes the key benefits of the TPM's major features.

FEATURE	BENEFITS WHEN USED ON A SYSTEM WITH A TPM
Platform Crypto Provider	• If the machine is compromised, the private key associated with the certificate cannot be copied off the device. • The TPM's dictionary attack mechanism protects PIN values to use a certificate.
Virtual Smart Card	• Achieve security similar to that of physical smart cards without deploying physical smart cards or card readers.
Windows Hello for Business	• Credentials provisioned on a device cannot be copied elsewhere. • Confirm a device's TPM before credentials are provisioned.
BitLocker Drive Encryption	• Multiple options are available for enterprises to protect data at rest while balancing security requirements with different device hardware.
Device Encryption	• With a Microsoft account and the right hardware, consumers' devices seamlessly benefit from data-at-rest protection.
Measured Boot	• A hardware root of trust contains boot measurements that help detect malware during remote attestation.
Health Attestation	• MDM solutions can easily perform remote attestation and evaluate client health before granting access to resources or cloud services such as Office 365.
Credential Guard	• Defense in depth increases so that even if malware has administrative rights on one machine, it is significantly more difficult to compromise additional machines in an organization.

Although some of the aforementioned features have additional hardware requirements (e.g., virtualization support), the TPM is a cornerstone of Windows 10 security. Microsoft and other industry stakeholders continue to improve the global standards associated with TPM and find more and more applications that use it to provide tangible benefits to customers. Microsoft has included support for most TPM features in its version of Windows for the Internet of Things (IoT) called Windows 10 IoT Core. IoT devices that might be deployed in insecure

physical locations and connected to cloud services like Azure IoT Hub for management can use the TPM in innovative ways to address their emerging security requirements.

TPM Group Policy settings

3/27/2021 • 9 minutes to read • Edit Online

Applies to

- Windows 10
- Windows Server 2016 and later

This topic describes the Trusted Platform Module (TPM) Services that can be controlled centrally by using Group Policy settings.

The Group Policy settings for TPM services are located at:

Computer Configuration\Administrative Templates\System\Trusted Platform Module Services

The following Group Policy settings were introduced in Windows 10.

Configure the level of TPM owner authorization information available to the operating system

> **IMPORTANT**
> Beginning with Windows 10 version 1607 and Windows Server 2016, this policy setting is no longer used by Windows, but it continues to appear in GPEdit.msc for compatibility with previous versions. Beginning with Windows 10 version 1703, the default value is 5. This value is implemented during provisioning so that another Windows component can either delete it or take ownership of it, depending on the system configuration. For TPM 2.0, a value of 5 means keep the lockout authorization. For TPM 1.2, it means discard the Full TPM owner authorization and retain only the Delegated authorization.

This policy setting configured which TPM authorization values are stored in the registry of the local computer. Certain authorization values are required in order to allow Windows to perform certain actions.

TPM 1.2 VALUE	TPM 2.0 VALUE	PURPOSE	KEPT AT LEVEL 0?	KEPT AT LEVEL 2?	KEPT AT LEVEL 4?
OwnerAuthAdmin	StorageOwnerAuth	Create SRK	No	Yes	Yes
OwnerAuthEndorsement	EndorsementAuth	Create or use EK (1.2 only: Create AIK)	No	Yes	Yes
OwnerAuthFull	LockoutAuth	Reset/change Dictionary Attack Protection	No	No	Yes

There are three TPM owner authentication settings that are managed by the Windows operating system. You can choose a value of **Full**, **Delegate**, or **None**.

- **Full** This setting stores the full TPM owner authorization, the TPM administrative delegation blob, and the TPM user delegation blob in the local registry. With this setting, you can use the TPM without

requiring remote or external storage of the TPM owner authorization value. This setting is appropriate for scenarios that do not require you to reset the TPM anti-hammering logic or change the TPM owner authorization value. Some TPM-based applications may require that this setting is changed before features that depend on the TPM anti-hammering logic can be used. Full owner authorization in TPM 1.2 is similar to lockout authorization in TPM 2.0. Owner authorization has a different meaning for TPM 2.0.

- **Delegated** This setting stores only the TPM administrative delegation blob and the TPM user delegation blob in the local registry. This setting is appropriate for use with TPM-based applications that depend on the TPM antihammering logic. This is the default setting in Windows prior to version 1703.
- **None** This setting provides compatibility with previous operating systems and applications. You can also use it for scenarios when TPM owner authorization cannot be stored locally. Using this setting might cause issues with some TPM-based applications.

> **NOTE**
> If the operating system managed TPM authentication setting is changed from **Full** to **Delegated**, the full TPM owner authorization value will be regenerated, and any copies of the previously set TPM owner authorization value will be invalid.

Registry information

Registry key: HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\TPM

DWORD: OSManagedAuthLevel

The following table shows the TPM owner authorization values in the registry.

VALUE DATA	SETTING
0	None
2	Delegated
4	Full

If you enable this policy setting, the Windows operating system will store the TPM owner authorization in the registry of the local computer according to the TPM authentication setting you choose.

On Windows 10 prior to version 1607, if you disable or do not configure this policy setting, and the **Turn on TPM backup to Active Directory Domain Services** policy setting is also disabled or not configured, the default setting is to store the full TPM authorization value in the local registry. If this policy is disabled or not configured, and the **Turn on TPM backup to Active Directory Domain Services** policy setting is enabled, only the administrative delegation and the user delegation blobs are stored in the local registry.

Standard User Lockout Duration

This policy setting allows you to manage the duration in minutes for counting standard user authorization failures for Trusted Platform Module (TPM) commands requiring authorization. An authorization failure occurs each time a standard user sends a command to the TPM and receives an error response that indicates an authorization failure occurred. Authorization failures that are older than the duration you set are ignored. If the number of TPM commands with an authorization failure within the lockout duration equals a threshold, a standard user is prevented from sending commands that require authorization to the TPM.

The TPM is designed to protect itself against password guessing attacks by entering a hardware lockout mode when it receives too many commands with an incorrect authorization value. When the TPM enters a lockout

mode, it is global for all users (including administrators) and for Windows features such as BitLocker Drive Encryption.

This setting helps administrators prevent the TPM hardware from entering a lockout mode by slowing the speed at which standard users can send commands that require authorization to the TPM.

For each standard user, two thresholds apply. Exceeding either threshold prevents the user from sending a command that requires authorization to the TPM. Use the following policy settings to set the lockout duration:

- Standard User Individual Lockout Threshold This value is the maximum number of authorization failures that each standard user can have before the user is not allowed to send commands that require authorization to the TPM.

- Standard User Total Lockout Threshold This value is the maximum total number of authorization failures that all standard users can have before all standard users are not allowed to send commands that require authorization to the TPM.

An administrator with the TPM owner password can fully reset the TPM's hardware lockout logic by using the Windows Defender Security Center. Each time an administrator resets the TPM's hardware lockout logic, all prior standard user TPM authorization failures are ignored. This allows standard users to immediately use the TPM normally.

If you do not configure this policy setting, a default value of 480 minutes (8 hours) is used.

Standard User Individual Lockout Threshold

This policy setting allows you to manage the maximum number of authorization failures for each standard user for the Trusted Platform Module (TPM). This value is the maximum number of authorization failures that each standard user can have before the user is not allowed to send commands that require authorization to the TPM. If the number of authorization failures for the user within the duration that is set for the **Standard User Lockout Duration** policy setting equals this value, the standard user is prevented from sending commands that require authorization to the Trusted Platform Module (TPM).

This setting helps administrators prevent the TPM hardware from entering a lockout mode by slowing the speed at which standard users can send commands that require authorization to the TPM.

An authorization failure occurs each time a standard user sends a command to the TPM and receives an error response indicating an authorization failure occurred. Authorization failures older than the duration are ignored.

An administrator with the TPM owner password can fully reset the TPM's hardware lockout logic by using the Windows Defender Security Center. Each time an administrator resets the TPM's hardware lockout logic, all prior standard user TPM authorization failures are ignored. This allows standard users to immediately use the TPM normally.

If you do not configure this policy setting, a default value of 4 is used. A value of zero means that the operating system will not allow standard users to send commands to the TPM, which might cause an authorization failure.

Standard User Total Lockout Threshold

This policy setting allows you to manage the maximum number of authorization failures for all standard users for the Trusted Platform Module (TPM). If the total number of authorization failures for all standard users within the duration that is set for the **Standard User Lockout Duration** policy equals this value, all standard users are prevented from sending commands that require authorization to the Trusted Platform Module (TPM).

This setting helps administrators prevent the TPM hardware from entering a lockout mode because it slows the speed standard users can send commands requiring authorization to the TPM.

An authorization failure occurs each time a standard user sends a command to the TPM and receives an error response indicating an authorization failure occurred. Authorization failures older than the duration are ignored.

An administrator with the TPM owner password can fully reset the TPM's hardware lockout logic by using the Windows Defender Security Center. Each time an administrator resets the TPM's hardware lockout logic, all prior standard user TPM authorization failures are ignored. This allows standard users to immediately use the TPM normally.

If you do not configure this policy setting, a default value of 9 is used. A value of zero means that the operating system will not allow standard users to send commands to the TPM, which might cause an authorization failure.

Configure the system to use legacy Dictionary Attack Prevention Parameters setting for TPM 2.0

Introduced in Windows 10, version 1703, this policy setting configures the TPM to use the Dictionary Attack Prevention Parameters (lockout threshold and recovery time) to the values that were used for Windows 10 Version 1607 and below.

IMPORTANT

Setting this policy will take effect only if:

- The TPM was originally prepared using a version of Windows after Windows 10 Version 1607
- The system has a TPM 2.0.

NOTE

Enabling this policy will only take effect after the TPM maintenance task runs (which typically happens after a system restart). Once this policy has been enabled on a system and has taken effect (after a system restart), disabling it will have no impact and the system's TPM will remain configured using the legacy Dictionary Attack Prevention parameters, regardless of the value of this group policy. The only ways for the disabled setting of this policy to take effect on a system where it was once enabled are to either:

- Disable it from group policy
- Clear the TPM on the system

TPM Group Policy settings in the Windows Security app

You can change what users see about TPM in the Windows Security app. The Group Policy settings for the TPM area in the Windows Security app are located at:

Computer Configuration\Administrative Templates\Windows Components\Windows Security\Device security

Disable the Clear TPM button

If you don't want users to be able to click the **Clear TPM** button in the Windows Security app, you can disable it with this Group Policy setting. Select **Enabled** to make the **Clear TPM** button unavailable for use.

Hide the TPM Firmware Update recommendation

If you don't want users to see the recommendation to update TPM firmware, you can disable it with this setting. Select **Enabled** to prevent users from seeing a recommendation to update their TPM firmware when a vulnerable firmware is detected.

Related topics

- Trusted Platform Module
- TPM Cmdlets in Windows PowerShell
- Prepare your organization for BitLocker: Planning and Policies - TPM configurations

Back up the TPM recovery information to AD DS

3/26/2021 • 2 minutes to read • Edit Online

Applies to

- Windows 10, version 1511
- Windows 10, version 1507

Does not apply to

- Windows 10, version 1607 or later

With Windows 10, versions 1511 and 1507, you can back up a computer's Trusted Platform Module (TPM) information to Active Directory Domain Services (AD DS). By doing this, you can use AD DS to administer the TPM from a remote computer. The procedure is the same as it was for Windows 8.1. For more information, see Backup the TPM Recovery Information to AD DS.

Related topics

- Trusted Platform Module (list of topics)
- TPM Group Policy settings

Troubleshoot the TPM

3/26/2021 • 7 minutes to read • Edit Online

Applies to

- Windows 10
- Windows Server 2016

This topic provides information for the IT professional to troubleshoot the Trusted Platform Module (TPM):

- Troubleshoot TPM initialization
- Clear all the keys from the TPM

With TPM 1.2 and Windows 10, version 1507 or 1511, you can also take the following actions:

- Turn on or turn off the TPM

For information about the TPM cmdlets, see TPM Cmdlets in Windows PowerShell.

About TPM initialization and ownership

Starting with Windows 10, the operating system automatically initializes and takes ownership of the TPM. This is a change from previous operating systems, where you would initialize the TPM and create an owner password.

Troubleshoot TPM initialization

If you find that Windows is not able to initialize the TPM automatically, review the following information:

- You can try clearing the TPM to the factory default values and allowing Windows to re-initialize it. For important precautions for this process, and instructions for completing it, see Clear all the keys from the TPM, later in this topic.
- If the TPM is a TPM 2.0 and is not detected by Windows, verify that your computer hardware contains a Unified Extensible Firmware Interface (UEFI) that is Trusted Computing Group-compliant. Also, ensure that in the UEFI settings, the TPM has not been disabled or hidden from the operating system.
- If you have TPM 1.2 with Windows 10, version 1507 or 1511, the TPM might be turned off, and need to be turned back on, as described in Turn on the TPM. When it is turned back on, Windows will re-initialize it.
- If you are attempting to set up BitLocker with the TPM, check which TPM driver is installed on the computer. We recommend always using one of the TPM drivers that is provided by Microsoft and is protected with BitLocker. If a non-Microsoft TPM driver is installed, it may prevent the default TPM driver from loading and cause BitLocker to report that a TPM is not present on the computer. If you have a non-Microsoft driver installed, remove it and then allow the operating system to initialize the TPM.

Troubleshoot network connection issues for Windows 10, versions 1507 and 1511

If you have Windows 10, version 1507 or 1511, the initialization of the TPM cannot complete when your computer has network connection issues and both of the following conditions exist:

- An administrator has configured your computer to require that TPM recovery information be saved in Active Directory Domain Services (AD DS). This requirement can be configured through Group Policy.

- A domain controller cannot be reached. This can occur on a computer that is currently disconnected from the network, separated from the domain by a firewall, or experiencing a network component failure (such as an unplugged cable or a faulty network adapter).

If these issues occur, an error message appears, and you cannot complete the initialization process. To avoid this issue, allow Windows to initialize the TPM while you are connected to the corporate network and you can contact a domain controller.

Troubleshoot systems with multiple TPMs

Some systems may have multiple TPMs and the active TPM may be toggled in UEFI. Windows 10 does not support this behavior. If you switch TPMs, Windows might not properly detect or interact with the new TPM. If you plan to switch TPMs you should toggle to the new TPM, clear it, and reinstall Windows. For more information, see Clear all the keys from the TPM, later in this topic.

For example, toggling TPMs will cause BitLocker to enter recovery mode. We strongly recommend that, on systems with two TPMs, one TPM is selected to be used and the selection is not changed.

Clear all the keys from the TPM

You can use the Windows Defender Security Center app to clear the TPM as a troubleshooting step, or as a final preparation before a clean installation of a new operating system. Preparing for a clean installation in this way helps ensure that the new operating system can fully deploy any TPM-based functionality that it includes, such as attestation. However, even if the TPM is not cleared before a new operating system is installed, most TPM functionality will probably work correctly.

Clearing the TPM resets it to an unowned state. After you clear the TPM, the Windows 10 operating system will automatically re-initialize it and take ownership again.

> **WARNING**
>
> Clearing the TPM can result in data loss. For more information, see the next section, “Precautions to take before clearing the TPM.”

Precautions to take before clearing the TPM

Clearing the TPM can result in data loss. To protect against such loss, review the following precautions:

- Clearing the TPM causes you to lose all created keys associated with the TPM, and data protected by those keys, such as a virtual smart card or a login PIN. Make sure that you have a backup and recovery method for any data that is protected or encrypted by the TPM.

- Do not clear the TPM on a device you do not own, such as a work or school PC, without being instructed to do so by your IT administrator.

- If you want to temporarily suspend TPM operations and you have TPM 1.2 with Windows 10, version 1507 or 1511, you can turn off the TPM. For more information, see Turn off the TPM, later in this topic.

- Always use functionality in the operating system (such as TPM.msc) to the clear the TPM. Do not clear the TPM directly from UEFI.

- Because your TPM security hardware is a physical part of your computer, before clearing the TPM, you might want to read the manuals or instructions that came with your computer, or search the manufacturer's website.

Membership in the local Administrators group, or equivalent, is the minimum required to complete this procedure.

To clear the TPM

1. Open the Windows Defender Security Center app.
2. Click **Device security**.
3. Click **Security processor details**.
4. Click **Security processor troubleshooting**.
5. Click **Clear TPM**.
6. You will be prompted to restart the computer. During the restart, you might be prompted by the UEFI to press a button to confirm that you wish to clear the TPM.
7. After the PC restarts, your TPM will be automatically prepared for use by Windows 10.

Turn on or turn off the TPM (available only with TPM 1.2 with Windows 10, version 1507 or 1511)

Normally, the TPM is turned on as part of the TPM initialization process. You do not normally need to turn the TPM on or off. However, if necessary you can do so by using the TPM MMC.

Turn on the TPM

If you want to use the TPM after you have turned it off, you can use the following procedure to turn on the TPM.

To turn on the TPM (TPM 1.2 with Windows 10, version 1507 or 1511 only)

1. Open the TPM MMC (tpm.msc).
2. In the **Action** pane, click **Turn TPM On** to display the **Turn on the TPM Security Hardware** page. Read the instructions on this page.
3. Click **Shutdown** (or **Restart**), and then follow the UEFI screen prompts.

   After the computer restarts, but before you sign in to Windows, you will be prompted to accept the reconfiguration of the TPM. This ensures that the user has physical access to the computer and that malicious software is not attempting to make changes to the TPM.

Turn off the TPM

If you want to stop using the services that are provided by the TPM, you can use the TPM MMC to turn off the TPM.

To turn off the TPM (TPM 1.2 with Windows 10, version 1507 or 1511 only)

1. Open the TPM MMC (tpm.msc).
2. In the **Action** pane, click **Turn TPM Off** to display the **Turn off the TPM security hardware** page.
3. In the **Turn off the TPM security hardware** dialog box, select a method to enter your owner password and turning off the TPM:

   - If you saved your TPM owner password on a removable storage device, insert it, and then click **I have the owner password file**. In the **Select backup file with the TPM owner password** dialog box, click **Browse** to locate the .tpm file that is saved on your removable storage device, click **Open**, and then click **Turn TPM Off**.
   - If you do not have the removable storage device with your saved TPM owner password, click **I want to enter the password**. In the **Type your TPM owner password** dialog box, type your password (including hyphens), and then click **Turn TPM Off**.

- If you did not save your TPM owner password or no longer know it, click **I do not have the TPM owner password**, and follow the instructions that are provided in the dialog box and subsequent UEFI screens to turn off the TPM without entering the password.

Use the TPM cmdlets

You can manage the TPM using Windows PowerShell. For details, see TPM Cmdlets in Windows PowerShell.

Related topics

- Trusted Platform Module (list of topics)

Understanding PCR banks on TPM 2.0 devices

3/5/2021 • 4 minutes to read • Edit Online

Applies to

- Windows 10
- Windows Server 2016

For steps on how to switch PCR banks on TPM 2.0 devices on your PC, you should contact your OEM or UEFI vendor. This topic provides background about what happens when you switch PCR banks on TPM 2.0 devices.

A Platform Configuration Register (PCR) is a memory location in the TPM that has some unique properties. The size of the value that can be stored in a PCR is determined by the size of a digest generated by an associated hashing algorithm. A SHA-1 PCR can store 20 bytes – the size of a SHA-1 digest. Multiple PCRs associated with the same hashing algorithm are referred to as a PCR bank.

To store a new value in a PCR, the existing value is extended with a new value as follows: PCR[N] = HASHalg(PCR[N] || ArgumentOfExtend)

The existing value is concatenated with the argument of the TPM Extend operation. The resulting concatenation is then used as input to the associated hashing algorithm, which computes a digest of the input. This computed digest becomes the new value of the PCR.

The TCG PC Client Platform TPM Profile Specification defines the inclusion of at least one PCR bank with 24 registers. The only way to reset the first 16 PCRs is to reset the TPM itself. This restriction helps ensure that the value of those PCRs can only be modified via the TPM Extend operation.

Some TPM PCRs are used as checksums of log events. The log events are extended in the TPM as the events occur. Later, an auditor can validate the logs by computing the expected PCR values from the log and comparing them to the PCR values of the TPM. Since the first 16 TPM PCRs cannot be modified arbitrarily, a match between an expected PCR value in that range and the actual TPM PCR value provides assurance of an unmodified log.

How does Windows 10 use PCRs?

To bind the use of a TPM based key to a certain state of the PC, the key can be sealed to an expected set of PCR values. For instance, PCRs 0 through 7 have a well-defined value after the boot process – when the OS is loaded. When the hardware, firmware, or boot loader of the machine changes, the change can be detected in the PCR values. Windows 10 uses this capability to make certain cryptographic keys only available at certain times during the boot process. For instance, the BitLocker key can be used at a certain point in the boot, but not before or after.

It is important to note that this binding to PCR values also includes the hashing algorithm used for the PCR. For instance, a key can be bound to a specific value of the SHA-1 PCR[12], if using SHA-256 PCR banks, even with the same system configuration. Otherwise, the PCR values will not match.

What happens when PCR banks are switched?

When the PCR banks are switched, the algorithm used to compute the hashed values stored in the PCRs during extend operations is changed. Each hash algorithm will return a different cryptographic signature for the same inputs.

As a result, if the currently used PCR bank is switched all keys that have been bound to the previous PCR values will no longer work. For example, if you had a key bound to the SHA-1 value of PCR[12] and subsequently

changed the PCR banks to SHA-256, the banks wouldn't match, and you would be unable to use that key. The BitLocker key is secured using the PCR banks and Windows 10 will not be able to unseal it if the PCR banks are switched while BitLocker is enabled.

What can I do to switch PCRs when BitLocker is already active?

Before switching PCR banks you should suspend or disable BitLocker – or have your recovery key ready. For steps on how to switch PCR banks on your PC, you should contact your OEM or UEFI vendor.

How can I identify which PCR bank is being used?

A TPM can be configured to have multiple PCR banks active. When BIOS is performing measurements it will do so into all active PCR banks, depending on its capability to make these measurements. BIOS may chose to deactivate PCR banks that it does not support or "cap" PCR banks that it does not support by extending a separator. The following registry value identifies which PCR banks are active.

- Registry key: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\IntegrityServices
- DWORD: TPMActivePCRBanks
- Defines which PCR banks are currently active. (This value should be interpreted as a bitmap for which the bits are defined in the TCG Algorithm Registry Table 21 of Revision 1.27.)

Windows checks which PCR banks are active and supported by the BIOS. Windows also checks if the measured boot log supports measurements for all active PCR banks. Windows will prefer the use of the SHA-256 bank for measurements and will fall back to SHA1 PCR bank if one of the pre-conditions is not met.

You can identify which PCR bank is currently used by Windows by looking at the registry.

- Registry key: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\IntegrityServices
- DWORD: TPMDigestAlgID
- Algorithm ID of the PCR bank that Windows is currently using. (This value represents an algorithm identifier as defined in the TCG Algorithm Registry Table 3 of Revision 1.27.)

Windows only uses one PCR bank to continue boot measurements. All other active PCR banks will be extended with a separator to indicate that they are not used by Windows and measurements that appear to be from Windows should not be trusted.

Related topics

- Trusted Platform Module (list of topics)

TPM recommendations

6/28/2021 • 8 minutes to read • Edit Online

Applies to

- Windows 10
- Windows Server 2016

This topic provides recommendations for Trusted Platform Module (TPM) technology for Windows 10.

For a basic feature description of TPM, see the Trusted Platform Module Technology Overview.

TPM design and implementation

Traditionally, TPMs have been discrete chips soldered to a computer's motherboard. Such implementations allow the computer's original equipment manufacturer (OEM) to evaluate and certify the TPM separate from the rest of the system. Although discrete TPM implementations are still common, they can be problematic for integrated devices that are small or have low power consumption. Some newer TPM implementations integrate TPM functionality into the same chipset as other platform components while still providing logical separation similar to discrete TPM chips.

TPMs are passive: they receive commands and return responses. To realize the full benefit of a TPM, the OEM must carefully integrate system hardware and firmware with the TPM to send it commands and react to its responses. TPMs were originally designed to provide security and privacy benefits to a platform's owner and users, but newer versions can provide security and privacy benefits to the system hardware itself. Before it can be used for advanced scenarios, however, a TPM must be provisioned. Windows 10 automatically provisions a TPM, but if the user is planning to reinstall the operating system, he or she may need to clear the TPM before reinstalling so that Windows can take full advantage of the TPM.

The Trusted Computing Group (TCG) is the nonprofit organization that publishes and maintains the TPM specification. The TCG exists to develop, define, and promote vendor-neutral, global industry standards that support a hardware-based root of trust for interoperable trusted computing platforms. The TCG also publishes the TPM specification as the international standard ISO/IEC 11889, using the Publicly Available Specification Submission Process that the Joint Technical Committee 1 defines between the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC).

OEMs implement the TPM as a component in a trusted computing platform, such as a PC, tablet, or phone. Trusted computing platforms use the TPM to support privacy and security scenarios that software alone cannot achieve. For example, software alone cannot reliably report whether malware is present during the system startup process. The close integration between TPM and platform increases the transparency of the startup process and supports evaluating device health by enabling reliable measuring and reporting of the software that starts the device. Implementation of a TPM as part of a trusted computing platform provides a hardware root of trust—that is, it behaves in a trusted way. For example, if a key stored in a TPM has properties that disallow exporting the key, that key truly cannot leave the TPM.

The TCG designed the TPM as a low-cost, mass-market security solution that addresses the requirements of different customer segments. There are variations in the security properties of different TPM implementations just as there are variations in customer and regulatory requirements for different sectors. In public-sector procurement, for example, some governments have clearly defined security requirements for TPMs whereas others do not.

TPM 1.2 vs. 2.0 comparison

From an industry standard, Microsoft has been an industry leader in moving and standardizing on TPM 2.0, which has many key realized benefits across algorithms, crypto, hierarchy, root keys, authorization and NV RAM.

Why TPM 2.0?

TPM 2.0 products and systems have important security advantages over TPM 1.2, including:

- The TPM 1.2 spec only allows for the use of RSA and the SHA-1 hashing algorithm.
- For security reasons, some entities are moving away from SHA-1. Notably, NIST has required many federal agencies to move to SHA-256 as of 2014, and technology leaders, including Microsoft and Google have announced they will remove support for SHA-1 based signing or certificates in 2017.
- TPM 2.0 **enables greater crypto agility** by being more flexible with respect to cryptographic algorithms.
    - TPM 2.0 supports newer algorithms, which can improve drive signing and key generation performance. For the full list of supported algorithms, see the TCG Algorithm Registry. Some TPMs do not support all algorithms.
    - For the list of algorithms that Windows supports in the platform cryptographic storage provider, see CNG Cryptographic Algorithm Providers.
    - TPM 2.0 achieved ISO standardization (ISO/IEC 11889:2015).
    - Use of TPM 2.0 may help eliminate the need for OEMs to make exception to standard configurations for certain countries and regions.
- TPM 2.0 offers a more **consistent experience** across different implementations.
    - TPM 1.2 implementations vary in policy settings. This may result in support issues as lockout policies vary.
    - TPM 2.0 lockout policy is configured by Windows, ensuring a consistent dictionary attack protection guarantee.
- While TPM 1.2 parts are discrete silicon components which are typically soldered on the motherboard, TPM 2.0 is available as a **discrete (dTPM)** silicon component in a single semiconductor package, an **integrated** component incorporated in one or more semiconductor packages - alongside other logic units in the same package(s) - and as a **firmware (fTPM)** based component running in a trusted execution environment (TEE) on a general purpose SoC.

> **NOTE**
>
> TPM 2.0 is not supported in Legacy and CSM Modes of the BIOS. Devices with TPM 2.0 must have their BIOS mode configured as Native UEFI only. The Legacy and Compatibility Support Module (CSM) options must be disabled. For added security Enable the Secure Boot feature.
>
> Installed Operating System on hardware in legacy mode will stop the OS from booting when the BIOS mode is changed to UEFI. Use the tool MBR2GPT before changing the BIOS mode which will prepare the OS and the disk to support UEFI.

Discrete, Integrated or Firmware TPM?

There are three implementation options for TPMs:

- Discrete TPM chip as a separate component in its own semiconductor package

- Integrated TPM solution, using dedicated hardware integrated into one or more semiconductor packages alongside, but logically separate from, other components
- Firmware TPM solution, running the TPM in firmware in a Trusted Execution mode of a general purpose computation unit

Windows uses any compatible TPM in the same way. Microsoft does not take a position on which way a TPM should be implemented and there is a wide ecosystem of available TPM solutions which should suit all needs.

Is there any importance for TPM for consumers?

For end consumers, TPM is behind the scenes but is still very relevant. TPM is used for Windows Hello, Windows Hello for Business and in the future, will be a component of many other key security features in Windows. TPM secures the PIN, helps encrypt passwords, and builds on our overall Windows 10 experience story for security as a critical pillar. Using Windows on a system with a TPM enables a deeper and broader level of security coverage.

TPM 2.0 Compliance for Windows 10

Windows 10 for desktop editions (Home, Pro, Enterprise, and Education)

- Since July 28, 2016, all new device models, lines or series (or if you are updating the hardware configuration of an existing model, line or series with a major update, such as CPU, graphic cards) must implement and enable by default TPM 2.0 (details in section 3.7 of the Minimum hardware requirements page). The requirement to enable TPM 2.0 only applies to the manufacturing of new devices. For TPM recommendations for specific Windows features, see TPM and Windows Features.

IoT Core

- TPM is optional on IoT Core.

Windows Server 2016

- TPM is optional for Windows Server SKUs unless the SKU meets the additional qualification (AQ) criteria for the Host Guardian Services scenario in which case TPM 2.0 is required.

TPM and Windows Features

The following table defines which Windows features require TPM support.

WINDOWS FEATURES	TPM REQUIRED	SUPPORTS TPM 1.2	SUPPORTS TPM 2.0	DETAILS
Measured Boot	Yes	Yes	Yes	Measured Boot requires TPM 1.2 or 2.0 and UEFI Secure Boot. TPM 2.0 is recommended since it supports newer cryptographic algorithms. TPM 1.2 only supports the SHA-1 algorithm which is being deprecated.

WINDOWS FEATURES	TPM REQUIRED	SUPPORTS TPM 1.2	SUPPORTS TPM 2.0	DETAILS
BitLocker	No	Yes	Yes	TPM 1.2 or 2.0 are supported but TPM 2.0 is recommended. Automatic Device Encryption requires Modern Standby including TPM 2.0 support
Device Encryption	Yes	N/A	Yes	Device Encryption requires Modern Standby/Connected Standby certification, which requires TPM 2.0.
Windows Defender Application Control (Device Guard)	No	Yes	Yes	
Windows Defender System Guard (DRTM)	Yes	No	Yes	TPM 2.0 and UEFI firmware is required.
Credential Guard	No	Yes	Yes	Windows 10, version 1507 (End of Life as of May 2017) only supported TPM 2.0 for Credential Guard. Beginning with Windows 10, version 1511, TPM 1.2 and 2.0 are supported. Paired with Windows Defender System Guard, TPM 2.0 provides enhanced security for Credential Guard. Windows 11 requires TPM 2.0 by default to facilitate easier enablement of this enhanced security for customers.
Device Health Attestation	Yes	Yes	Yes	TPM 2.0 is recommended since it supports newer cryptographic algorithms. TPM 1.2 only supports the SHA-1 algorithm which is being deprecated.

WINDOWS FEATURES	TPM REQUIRED	SUPPORTS TPM 1.2	SUPPORTS TPM 2.0	DETAILS
Windows Hello/Windows Hello for Business	No	Yes	Yes	Azure AD join supports both versions of TPM, but requires TPM with keyed-hash message authentication code (HMAC) and Endorsement Key (EK) certificate for key attestation support. TPM 2.0 is recommended over TPM 1.2 for better performance and security. Windows Hello as a FIDO platform authenticator will take advantage of TPM 2.0 for key storage.
UEFI Secure Boot	No	Yes	Yes	
TPM Platform Crypto Provider Key Storage Provider	Yes	Yes	Yes	
Virtual Smart Card	Yes	Yes	Yes	
Certificate storage	No	Yes	Yes	TPM is only required when the certificate is stored in the TPM.
Autopilot	No	N/A	Yes	If you intend to deploy a scenario which requires TPM (such as white glove and self-deploying mode), then TPM 2.0 and UEFI firmware are required.
SecureBIO	Yes	No	Yes	TPM 2.0 and UEFI firmware is required.

OEM Status on TPM 2.0 system availability and certified parts

Government customers and enterprise customers in regulated industries may have acquisition standards that require use of common certified TPM parts. As a result, OEMs, who provide the devices, may be required to use only certified TPM components on their commercial class systems. For more information, contact your OEM or hardware vendor.

Related topics

- Trusted Platform Module (list of topics)

EXHIBIT D

Microsoft

Windows 10

Minimum Hardware Requirements

December 2019

Copyright

Contents

Section 3.0 - Minimum hardware requirements for Windows 10 for desktop editions

This section provides detailed hardware requirements that apply to any device that runs Windows 10 for desktop editions. See Table 2 for the list of devices that can run Windows 10 for desktop editions. For additional component requirements that may also apply, see section 6.0.

Note Throughout this specification, all requirements for Windows 10 for desktop editions also apply to Windows 10 Enterprise.

3.1 Processor

Devices that run Windows 10 for desktop editions require a 1 GHz or faster processor or SoC that meets the following requirements:

- Compatible with the x86* or x64 instruction set.
- Supports PAE, NX and SSE2.
- Supports CMPXCHG16b, LAHF/SAHF, and PrefetchW for 64-bit OS installation

* Beginning with Windows 10, version 2004, all new Windows 10 systems will be required to use 64-bit builds and Microsoft will no longer release 32-bit builds for OEM distribution. This does not impact 32-bit customer systems that are manufactured with earlier versions of Windows 10; Microsoft remains committed to providing feature and security updates on these devices, including continued 32-bit media availability in non-OEM channels to support various upgrade installation scenarios.

3.2 Memory

Devices that run Windows 10 for desktop editions must meet the RAM requirements shown in Table 11.

Table 11: RAM requirements for devices that run Windows 10 for desktop editions

OS architecture	RAM requirement
32-bit	>= 1 GB
64-bit	>= 2 GB

3.3 Storage

3.3.1 Storage device size

Devices that run Windows 10 for desktop editions must include a storage device that meets the size requirements shown in Table 12.

Table 12: Storage size requirements for devices that run Windows 10 for desktop editions

OS version	OS architecture	Storage capacity
Windows 10, version 1809 and prior	32-bit	16 GB or greater
	64-bit	20 GB or greater
Windows 10, version 1903	32-bit and 64-bit	32 GB or greater
Windows 10 IoT Enterprise, version 1903 and prior	32-bit	16GB or greater
	64-bit	20 GB or greater

3.3.2 Storage controller

Storage controllers used in devices that run Windows 10 for desktop editions must meet the following requirements:

- Storage controllers must support booting using the Extensible Firmware Interface (EFI) and implement device paths as defined in EDD-3.
- Storage host controllers and adapters must meet the requirements for the device protocol used and any requirements related to the device storage bus type.
- Bus-attached controllers must implement the correct class/subclass code as specified in the PCI Codes and Assignments v1.6 specification.

3.4 Display and graphics

3.4.1 Resolution, bit depth, and size

Windows 10 for desktop editions supports a minimum display resolution of SVGA (800 x 600) with a depth of 32 bits on each output simultaneously, and a minimum diagonal display size for the primary display of 7-inches or larger. Tablets, 2-in-1s, and laptops that run Windows 10 for desktop editions must include a display that meets the minimum requirements listed earlier. A display is optional for desktop PC's that run Windows 10 for desktop editions.

3.4.2 Graphics

Devices that run Windows 10 for desktop editions must include a GPU that supports DirectX 9 or later.

3.5 Networking

Devices that run Windows 10 for desktop editions must include at least one network connectivity option, such as Wi-Fi or an Ethernet adapter.

3.6 Hardware buttons

Table 13 lists the required, optional, and not supported hardware buttons for devices that run Windows 10 for desktop editions. All other buttons not included in this table, including custom hardware buttons specified by the OEM, are optional. See section 6.6 for additional requirements about hardware button behavior.

Table 13: Button implementation requirements for Windows 10 for desktop editions

Device type	Power button	Volume Up/ Volume Down button	Start button	Back/Search button	Camera button	Rotation lock button
Tablets	Required	Required	Optional[8]	Not supported	Not supported	Optional
Other devices	Required	Required for devices with detachable keyboards. Optional for all other devices	Optional[8]	Not supported	Not supported	Optional

[8] A software-rendered Start button is always available through the OS.

3.7 Trusted Platform Module (TPM)

As of July 28, 2016, all new device models, lines or series must implement and be in compliance with the International Standard ISO/IEC 11889:2015 or the Trusted Computing Group TPM 2.0 Library and a component which implements the TPM 2.0 must be present and enabled by default from this effective date.

The following requirements must be met:

- All TPM configurations must comply with local laws and regulations.
- Firmware-based components that implement TPM capabilities must implement version 2.0 of the TPM specification.
- An EK certificate must either be pre-provisioned to the TPM by the hardware vendor or be capable of being retrieved by the device during the first boot experience.
- It must ship with SHA-256 PCR banks and implement PCRs 0 through 23 for SHA-256. Note that it is acceptable to ship TPMs with a single switchable PCR bank that can be utilized for SHA-256 measurements.
- It must support TPM2_HMAC command.

A UEFI firmware option to turn off the TPM is not required. OEM systems for special purpose commercial systems, custom order, and customer systems with a custom image are not required to ship with a TPM support enabled.

For detailed TPM information, see Trusted Platform Module topic on TechNet and for TPM 1.2 and 2.0 version comparisons, please reference this article here.

Section 4.0 - Minimum hardware requirements for Windows Server 2016

This section provides detailed hardware requirements that apply only to computers that run Windows Server 2016. For additional component requirements that may also apply, see section 6.0.

4.1 Processor

Computers that run Windows Server 2016 require a 64-bit 1.4 GHz or faster processor or SoC that meets the following requirements:

EXHIBIT E

bcrypt.h header

01/11/2019 • 5 minutes to read

In this article

This header is used by Security and Identity. For more information, see:

- Security and Identity

bcrypt.h contains the following programming interfaces:

Functions

BCRYPT_INIT_AUTH_MODE_INFO

Initializes a BCRYPT_AUTHENTICATED_CIPHER_MODE_INFO structure for use in calls to BCryptEncrypt and BCryptDecrypt functions.

BCryptAddContextFunction

Adds a cryptographic function to the list of functions that are supported by an existing CNG context.

BCryptCloseAlgorithmProvider

Closes an algorithm provider.

BCryptConfigureContext

Sets the configuration information for an existing CNG context.

BCryptConfigureContextFunction

Sets the configuration information for the cryptographic function of an existing CNG context.

BCryptCreateContext

Creates a new CNG configuration context.

BCryptCreateHash

Called to create a hash or Message Authentication Code (MAC) object.

BCryptCreateMultiHash

The BCryptCreateMultiHash function creates a multi-hash state that allows for the parallel computation of multiple hash operations.

BCryptDecrypt

Decrypts a block of data.

BCryptDeleteContext

Deletes an existing CNG configuration context.

BCryptDeriveKey

Derives a key from a secret agreement value.

BCryptDeriveKeyCapi

Derives a key from a hash value.

BCryptDeriveKeyPBKDF2

Derives a key from a hash value by using the PBKDF2 key derivation algorithm as defined by

RFC 2898.

BCryptDestroyHash

Destroys a hash or Message Authentication Code (MAC) object.

BCryptDestroyKey

Destroys a key.

BCryptDestroySecret

Destroys a secret agreement handle that was created by using the BCryptSecretAgreement function.

BCryptDuplicateHash

Duplicates an existing hash or Message Authentication Code (MAC) object.

BCryptDuplicateKey

Creates a duplicate of a symmetric key.

BCryptEncrypt

Encrypts a block of data.

BCryptEnumAlgorithms

Gets a list of the registered algorithm identifiers.

BCryptEnumContextFunctionProviders

Obtains the providers for the cryptographic functions for a context in the specified configuration table.

BCryptEnumContextFunctions

Obtains the cryptographic functions for a context in the specified configuration table.

BCryptEnumContexts

Obtains the identifiers of the contexts in the specified configuration table.

BCryptEnumProviders

Obtains all of the CNG providers that support a specified algorithm.

BCryptEnumRegisteredProviders

Retrieves information about the registered providers.

BCryptExportKey

Exports a key to a memory BLOB that can be persisted for later use.

BCryptFinalizeKeyPair

Completes a public/private key pair.

BCryptFinishHash

Retrieves the hash or Message Authentication Code (MAC) value for the data accumulated from prior calls to BCryptHashData.

BCryptFreeBuffer

Used to free memory that was allocated by one of the CNG functions.

BCryptGenerateKeyPair

Creates an empty public/private key pair.

BCryptGenerateSymmetricKey

Creates a key object for use with a symmetrical key encryption algorithm from a supplied key.

BCryptGenRandom

Generates a random number.

BCryptGetFipsAlgorithmMode

Determines whether Federal Information Processing Standard (FIPS) compliance is enabled.

BCryptGetProperty

Retrieves the value of a named property for a CNG object.

BCryptHash

Performs a single hash computation. This is a convenience function that wraps calls to BCryptCreateHash, BCryptHashData, BCryptFinishHash, and BCryptDestroyHash.

BCryptHashData

Performs a one way hash or Message Authentication Code (MAC) on a data buffer.

BCryptImportKey

Imports a symmetric key from a key BLOB.

BCryptImportKeyPair

Imports a public/private key pair from a key BLOB.

BCryptKeyDerivation

Derives a key without requiring a secret agreement.

BCryptOpenAlgorithmProvider

Loads and initializes a CNG provider.

BCryptProcessMultiOperations

The BCryptProcessMultiOperations function processes a sequence of operations on a multi-object state.

BCryptQueryContextConfiguration

Retrieves the current configuration for the specified CNG context.

BCryptQueryContextFunctionConfiguration

Obtains the cryptographic function configuration information for an existing CNG context.

BCryptQueryContextFunctionProperty

Obtains the value of a named property for a cryptographic function in an existing CNG context.

BCryptQueryProviderRegistration

Retrieves information about a CNG provider.

BCryptRegisterConfigChangeNotify

Creates a user mode CNG configuration change event handler.

BCryptRegisterConfigChangeNotify

BCryptRemoveContextFunction

Removes a cryptographic function from the list of functions that are supported by an existing CNG context.

BCryptResolveProviders

Obtains a collection of all of the providers that meet the specified criteria.

BCryptSecretAgreement

Creates a secret agreement value from a private and a public key.

BCryptSetContextFunctionProperty

Sets the value of a named property for a cryptographic function in an existing CNG context.

BCryptSetProperty

Sets the value of a named property for a CNG object.

BCryptSignHash

Creates a signature of a hash value.

BCryptUnregisterConfigChangeNotify

Removes a user mode CNG configuration change event handler that was created by using the BCryptRegisterConfigChangeNotify(HANDLE*) function.

BCryptUnregisterConfigChangeNotify

BCryptVerifySignature

Verifies that the specified signature matches the specified hash.

Structures

BCRYPT_ALGORITHM_IDENTIFIER

Is used with the BCryptEnumAlgorithms function to contain a cryptographic algorithm identifier.

BCRYPT_AUTHENTICATED_CIPHER_MODE_INFO

Used with the BCryptEncrypt and BCryptDecrypt functions to contain additional information related to authenticated cipher modes.

BCRYPT_DH_KEY_BLOB

Used as a header for a Diffie-Hellman public key or private key BLOB in memory.

BCRYPT_DH_PARAMETER_HEADER

Used to contain parameter header information for a Diffie-Hellman key.

BCRYPT_DSA_KEY_BLOB

Used as a header for a Digital Signature Algorithm (DSA) public key or private key BLOB in memory.

BCRYPT_DSA_KEY_BLOB_V2

Used as a header for a Digital Signature Algorithm (DSA) public key or private key BLOB in memory.

BCRYPT_DSA_PARAMETER_HEADER

Used to contain parameter header information for a Digital Signature Algorithm (DSA) key.

BCRYPT_DSA_PARAMETER_HEADER_V2

Contains parameter header information for a Digital Signature Algorithm (DSA) key.

BCRYPT_ECCKEY_BLOB

Used as a header for an elliptic curve public key or private key BLOB in memory.

BCRYPT_INTERFACE_VERSION

Contains version information for a programmatic interface for a CNG provider.

BCRYPT_KEY_BLOB

Is the base structure for all CNG key BLOBs.

BCRYPT_KEY_DATA_BLOB_HEADER

Used to contain information about a key data BLOB.

BCRYPT_KEY_LENGTHS_STRUCT

Defines the range of key sizes that are supported by the provider.

BCRYPT_MULTI_HASH_OPERATION

A BCRYPT_MULTI_HASH_OPERATION structure defines a single operation in a multi-hash operation.

BCRYPT_MULTI_OBJECT_LENGTH_STRUCT

The BCRYPT_MULTI_OBJECT_LENGTH_STRUCT structure contains information to determine the size of the pbHashObject buffer for the BCryptCreateMultiHash function.

BCRYPT_OAEP_PADDING_INFO

Used to provide options for the Optimal Asymmetric Encryption Padding (OAEP) scheme.

BCRYPT_OID

Contains information about a DER-encoded object identifier (OID).

BCRYPT_OID_LIST

Used to contain a collection of BCRYPT_OID structures. Use this structure with the BCRYPT_HASH_OID_LIST property to retrieve the list of hashing object identifiers (OIDs) that have been encoded by using Distinguished Encoding Rules (DER) encoding.

BCRYPT_PKCS1_PADDING_INFO

Used to provide options for the PKCS

BCRYPT_PROVIDER_NAME

Contains the name of a CNG provider.

BCRYPT_PSS_PADDING_INFO

Used to provide options for the Probabilistic Signature Scheme (PSS) padding scheme.

BCRYPT_RSAKEY_BLOB

Used as a header for an RSA public key or private key BLOB in memory.

CRYPT_CONTEXT_CONFIG

Contains configuration information for a CNG context.

CRYPT_CONTEXT_FUNCTION_CONFIG

Contains configuration information for a cryptographic function of a CNG context.

CRYPT_CONTEXT_FUNCTION_PROVIDERS

Contains a set of cryptographic function providers for a CNG configuration context.

CRYPT_CONTEXT_FUNCTIONS

Contains a set of cryptographic functions for a CNG configuration context.

CRYPT_CONTEXTS

Contains a set of CNG configuration context identifiers.

CRYPT_IMAGE_REF

Contains information about a CNG provider module.

CRYPT_IMAGE_REG

Contains image registration information about a CNG provider.

CRYPT_INTERFACE_REG

Used to contain information about the type of interface supported by a CNG provider.

CRYPT_PROPERTY_REF

Contains information about a CNG context property.

CRYPT_PROVIDER_REF

Contains information about a cryptographic interface that a provider supports.

CRYPT_PROVIDER_REFS

Contains a collection of provider references.

CRYPT_PROVIDER_REG

Used to contain registration information for a CNG provider.

CRYPT_PROVIDERS

Contains information about the registered CNG providers.

Enumerations

BCRYPT_HASH_OPERATION_TYPE

The BCRYPT_HASH_OPERATION_TYPE enumeration specifies the hash operation type.

BCRYPT_MULTI_OPERATION_TYPE

The BCRYPT_MULTI_OPERATION_TYPE enumeration specifies type of multi-operation that is passed to the BCryptProcessMultiOperations function.

DSAFIPSVERSION_ENUM

Contains FIPS version information.

HASHALGORITHM_ENUM

Specifies signing and hashing algorithms.

Is this page helpful?

Yes No

EXHIBIT F

ncrypt.h header

01/11/2019 • 2 minutes to read

In this article

This header is used by Security and Identity. For more information, see:

- Security and Identity

ncrypt.h contains the following programming interfaces:

Functions

NCryptCreateClaim	Creates a key attestation claim.
NCryptCreatePersistedKey	Creates a new key and stores it in the specified key storage provider.
NCryptDecrypt	Decrypts a block of encrypted data.
NCryptDeleteKey	Deletes a CNG key.

NCryptDeriveKey

Derives a key from a secret agreement value.

NCryptEncrypt

Encrypts a block of data.

NCryptEnumAlgorithms

Obtains the names of the algorithms that are supported by the specified key storage provider.

NCryptEnumKeys

Obtains the names of the keys that are stored by the provider.

NCryptEnumStorageProviders

Obtains the names of the registered key storage providers.

NCryptExportKey

Exports a CNG key to a memory BLOB.

NCryptFinalizeKey

Completes a CNG key storage key.

NCryptFreeBuffer

Releases a block of memory allocated by a CNG key storage provider.

NCryptFreeObject

Frees a CNG key storage object.

NCryptGetProperty

Retrieves the value of a named property for a key storage object.

NCryptImportKey

Imports a Cryptography API:_Next Generation (CNG) key from a memory BLOB.

NCryptIsAlgSupported

Determines if a CNG key storage provider supports a specific cryptographic algorithm.

NCryptIsKeyHandle

Determines if the specified handle is a CNG key handle.

NCryptKeyDerivation

Creates a key from another key by using the specified key derivation function.

NCryptNotifyChangeKey

Creates or removes a key change notification.

NCryptOpenKey

Opens a key that exists in the specified CNG key storage provider.

NCryptOpenStorageProvider

Loads and initializes a CNG key storage provider.

NCryptSecretAgreement

Creates a secret agreement value from a private and a public key.

NCryptSetProperty

Sets the value for a named property for a CNG key storage object.

NCryptSignHash

Creates a signature of a hash value.

NCryptTranslateHandle

Translates a CryptoAPI handle into a CNG key handle.

NCryptVerifyClaim

Verifies a key attestation claim.

NCryptVerifySignature

Verifies that the specified signature matches the specified hash.

Structures

NCRYPT_ALLOC_PARA

Enables you to specify custom functions that can be used to allocate and free data.

NCRYPT_KEY_BLOB_HEADER

Contains a key BLOB.

NCRYPT_SUPPORTED_LENGTHS

Used with the NCRYPT_LENGTHS_PROPERTY property to contain length information for a key.

NCRYPT_UI_POLICY

Used with the NCRYPT_UI_POLICY_PROPERTY property to contain strong key user interface information for a key.

NCryptAlgorithmName

Used to contain information about a CNG algorithm.

NCryptKeyName

Used to contain information about a CNG key.

NCryptProviderName

Used to contain the name of a CNG key storage provider.

Is this page helpful?

Yes No

EXHIBIT G

wincrypt.h header

01/11/2019 • 44 minutes to read

In this article

This header is used by Security and Identity. For more information, see:

- Security and Identity

wincrypt.h contains the following programming interfaces:

Functions

CertAddCertificateContextToStore

Adds a certificate context to the certificate store.

CertAddCertificateLinkToStore

Adds a link in a certificate store to a certificate context in a different store.

CertAddCRLContextToStore

Adds a certificate revocation list (CRL) context to the specified certificate store.

CertAddCRLLinkToStore

Adds a link in a store to a certificate revocation list (CRL) context in a different store.

CertAddCTLContextToStore

Adds a certificate trust list (CTL) context to a certificate store.

CertAddCTLLinkToStore

The CertAddCTLLinkToStore function adds a link in a store to a certificate trust list (CTL) context in a different store. Instead of creating and adding a duplicate of a CTL context, this function adds a link to the original CTL context.

CertAddEncodedCertificateToStore

Creates a certificate context from an encoded certificate and adds it to the certificate store.

CertAddEncodedCertificateToSystemStoreA

Opens the specified system store and adds the encoded certificate to it.

CertAddEncodedCertificateToSystemStoreW

Opens the specified system store and adds the encoded certificate to it.

CertAddEncodedCRLToStore

Creates a certificate revocation list (CRL) context from an encoded CRL and adds it to the certificate store.

CertAddEncodedCTLToStore

Creates a certificate trust list (CTL) context from an encoded CTL and adds it to the certificate store.

CertAddEnhancedKeyUsageIdentifier

The CertAddEnhancedKeyUsageIdentifier function adds a usage identifier object identifier (OID) to the enhanced key usage (EKU) extended property of the certificate.

CertAddRefServerOcspResponse

Increments the reference count for an HCERT_SERVER_OCSP_RESPONSE handle.

CertAddRefServerOcspResponseContext

Increments the reference count for a CERT_SERVER_OCSP_RESPONSE_CONTEXT structure.

CertAddSerializedElementToStore

Adds a serialized certificate, certificate revocation list (CRL), or certificate trust list (CTL) element to the store.

CertAddStoreToCollection

The CertAddStoreToCollection function adds a sibling certificate store to a collection certificate store.

CertAlgIdToOID

Converts a CryptoAPI algorithm identifier (ALG_ID) to an Abstract Syntax Notation One (ASN.1) object identifier (OID) string.

CertCloseServerOcspResponse

Closes an online certificate status protocol (OCSP) server response handle.

CertCloseStore

Closes a certificate store handle and reduces the reference count on the store.

CertCompareCertificate

Determines whether two certificates are identical by comparing the issuer name and serial number of the certificates.

CertCompareCertificateName

The CertCompareCertificateName function compares two certificate CERT_NAME_BLOB structures to determine whether they are identical. The CERT_NAME_BLOB structures are used

for the subject and the issuer of certificates.

CertCompareIntegerBlob

The CertCompareIntegerBlob function compares two integer BLOBs to determine whether they represent equal numeric values.

CertComparePublicKeyInfo

The CertComparePublicKeyInfo function compares two encoded public keys to determine whether they are identical.

CertControlStore

Allows an application to be notified when there is a difference between the contents of a cached store in use and the contents of that store as it is persisted to storage.

CertCreateCertificateChainEngine

The CertCreateCertificateChainEngine function creates a new, nondefault chain engine for an application.

CertCreateCertificateContext

Creates a certificate context from an encoded certificate. The created context is not persisted to a certificate store. The function makes a copy of the encoded certificate within the created context.

CertCreateContext

Creates the specified context from the encoded bytes. The context created does not include any extended properties.

CertCreateCRLContext

The CertCreateCRLContext function creates a certificate revocation list (CRL) context from an encoded CRL. The created context is not persisted to a certificate store. It makes a copy of the encoded CRL within the created context.

CertCreateCTLContext

The CertCreateCTLContext function creates a certificate trust list (CTL) context from an encoded CTL. The created context is not persisted to a certificate store. The function makes a copy of the encoded CTL within the created context.

CertCreateCTLEntryFromCertificateContextProperties

The CertCreateCTLEntryFromCertificateContextProperties function creates a certificate trust list (CTL) entry whose attributes are the properties of the certificate context. The SubjectIdentifier in the CTL entry is the SHA1 hash of the certificate.

CertCreateSelfSignCertificate

Builds a self-signed certificate and returns a pointer to a CERT_CONTEXT structure that represents the certificate.

CertDeleteCertificateFromStore

The CertDeleteCertificateFromStore function deletes the specified certificate context from the certificate store.

CertDeleteCRLFromStore

The CertDeleteCRLFromStore function deletes the specified certificate revocation list (CRL) context from the certificate store.

CertDeleteCTLFromStore

The CertDeleteCTLFromStore function deletes the specified certificate trust list (CTL) context from a certificate store.

CertDuplicateCertificateChain

The CertDuplicateCertificateChain function duplicates a pointer to a certificate chain by incrementing the chain's reference count.

CertDuplicateCertificateContext

Duplicates a certificate context by incrementing its reference count.

CertDuplicateCRLContext

The CertDuplicateCRLContext function duplicates a certificate revocation list (CRL) context by incrementing its reference count.

CertDuplicateCTLContext

The CertDuplicateCTLContext function duplicates a certificate trust list (CTL) context by incrementing its reference count.

CertDuplicateStore

Duplicates a store handle by incrementing the store's reference count.

CertEnumCertificateContextProperties

The CertEnumCertificateContextProperties function retrieves the first or next extended property associated with a certificate context.

CertEnumCertificatesInStore

Retrieves the first or next certificate in a certificate store. Used in a loop, this function can retrieve in sequence all certificates in a certificate store.

CertEnumCRLContextProperties

The CertEnumCRLContextProperties function retrieves the first or next extended property associated with a certificate revocation list (CRL) context.

CertEnumCRLsInStore

The CertEnumCRLsInStore function retrieves the first or next certificate revocation list (CRL) context in a certificate store. Used in a loop, this function can retrieve in sequence all CRL contexts in a certificate store.

CertEnumCTLContextProperties

The CertEnumCTLContextProperties function retrieves the first or next extended property associated with a certificate trust list (CTL) context. Used in a loop, this function can retrieve in sequence all extended properties associated with a CTL context.

CertEnumCTLsInStore

The CertEnumCTLsInStore function retrieves the first or next certificate trust list (CTL) context in a certificate store. Used in a loop, this function can retrieve in sequence all CTL contexts in a certificate store.

CertEnumPhysicalStore

The CertEnumPhysicalStore function retrieves the physical stores on a computer. The function calls the provided callback function for each physical store found.

CertEnumSubjectInSortedCTL

Retrieves the first or next TrustedSubject in a sorted certificate trust list (CTL).

CertEnumSystemStore

The CertEnumSystemStore function retrieves the system stores available. The function calls the provided callback function for each system store found.

CertEnumSystemStoreLocation

The CertEnumSystemStoreLocation function retrieves all of the system store locations. The function calls the provided callback function for each system store location found.

CertFindAttribute

The CertFindAttribute function finds the first attribute in the CRYPT_ATTRIBUTE array, as identified by its object identifier (OID).

CertFindCertificateInCRL

The CertFindCertificateInCRL function searches the certificate revocation list (CRL) for the specified certificate.

CertFindCertificateInStore

Finds the first or next certificate context in a certificate store that matches a search criteria established by the dwFindType and its associated pvFindPara.

CertFindChainInStore

Finds the first or next certificate in a store that meets the specified criteria.

CertFindCRLInStore

Finds the first or next certificate revocation list (CRL) context in a certificate store that matches a search criterion established by the dwFindType parameter and the associated pvFindPara parameter.

CertFindCTLInStore

Finds the first or next certificate trust list (CTL) context that matches search criteria established by the dwFindType and its associated pvFindPara.

CertFindExtension

The CertFindExtension function finds the first extension in the CERT_EXTENSION array, as identified by its object identifier (OID).

CertFindRDNAttr

The CertFindRDNAttr function finds the first RDN attribute identified by its object identifier (OID) in a list of the Relative Distinguished Names (RDN).

CertFindSubjectInCTL

The CertFindSubjectInCTL function attempts to find the specified subject in a certificate trust list (CTL).

CertFindSubjectInSortedCTL

The CertFindSubjectInSortedCTL function attempts to find the specified subject in a sorted certificate trust list (CTL).

CertFreeCertificateChain

The CertFreeCertificateChain function frees a certificate chain by reducing its reference count. If the reference count becomes zero, memory allocated for the chain is released.

CertFreeCertificateChainEngine

The CertFreeCertificateChainEngine function frees a certificate trust engine.

CertFreeCertificateChainList

Frees the array of pointers to chain contexts.

CertFreeCertificateContext

Frees a certificate context by decrementing its reference count. When the reference count goes to zero, CertFreeCertificateContext frees the memory used by a certificate context.

CertFreeCRLContext

Frees a certificate revocation list (CRL) context by decrementing its reference count.

CertFreeCTLContext

Frees a certificate trust list (CTL) context by decrementing its reference count.

CertFreeServerOcspResponseContext

Decrements the reference count for a CERT_SERVER_OCSP_RESPONSE_CONTEXT structure.

CertGetCertificateChain

Builds a certificate chain context starting from an end certificate and going back, if possible, to a trusted root certificate.

CertGetCertificateContextProperty

Retrieves the information contained in an extended property of a certificate context.

CertGetCRLContextProperty

Gets an extended property for the specified certificate revocation list (CRL) context.

CertGetCRLFromStore

Gets the first or next certificate revocation list (CRL) context from the certificate store for the specified issuer.

CertGetCTLContextProperty

Retrieves an extended property of a certificate trust list (CTL) context.

CertGetEnhancedKeyUsage

Returns information from the enhanced key usage (EKU) extension or the EKU extended property of a certificate.

CertGetIntendedKeyUsage

Acquires the intended key usage bytes from a certificate.

CertGetIssuerCertificateFromStore

Retrieves the certificate context from the certificate store for the first or next issuer of the specified subject certificate. The new Certificate Chain Verification Functions are recommended instead of the use of this function.

CertGetNameStringA

Obtains the subject or issuer name from a certificate CERT_CONTEXT structure and converts it to a null-terminated character string.

CertGetNameStringW

Obtains the subject or issuer name from a certificate CERT_CONTEXT structure and converts it to a null-terminated character string.

CertGetPublicKeyLength

The CertGetPublicKeyLength function acquires the bit length of public/private keys from a public key BLOB.

CertGetServerOcspResponseContext

Retrieves a non-blocking, time valid online certificate status protocol (OCSP) response context for the specified handle.

CertGetStoreProperty

Retrieves a store property.

CertGetSubjectCertificateFromStore

Returns from a certificate store a subject certificate context uniquely identified by its issuer and serial number.

CertGetValidUsages

Returns an array of usages that consist of the intersection of the valid usages for all certificates in an array of certificates.

CertIsRDNAttrsInCertificateName

The CertIsRDNAttrsInCertificateName function compares the attributes in the certificate name with the specified CERT_RDN to determine whether all attributes are included there.

CertIsStrongHashToSign

Determines whether the specified hash algorithm and the public key in the signing certificate can be used to perform strong signing.

CertIsValidCRLForCertificate

The CertIsValidCRLForCertificate function checks a CRL to find out if it is a CRL that would include a specific certificate if that certificate were revoked.

CertNameToStrA

Converts an encoded name in a CERT_NAME_BLOB structure to a null-terminated character string.

CertNameToStrW

Converts an encoded name in a CERT_NAME_BLOB structure to a null-terminated character string.

CertOIDToAlgId

Use the CryptFindOIDInfo function instead of this function because ALG_ID identifiers are no longer supported in CNG.

CertOpenServerOcspResponse

Opens a handle to an online certificate status protocol (OCSP) response associated with a server certificate chain.

CertOpenStore

Opens a certificate store by using a specified store provider type.

CertOpenSystemStoreA

Opens the most common system certificate store. To open certificate stores with more complex requirements, such as file-based or memory-based stores, use CertOpenStore.

CertOpenSystemStoreW

Opens the most common system certificate store. To open certificate stores with more complex requirements, such as file-based or memory-based stores, use CertOpenStore.

CertRDNValueToStrA

The CertRDNValueToStr function converts a name in a CERT_RDN_VALUE_BLOB to a null-terminated character string.

CertRDNValueToStrW

The CertRDNValueToStr function converts a name in a CERT_RDN_VALUE_BLOB to a null-terminated character string.

CertRegisterPhysicalStore

Adds a physical store to a registry system store collection.

CertRegisterSystemStore

Registers a system store.

CertRemoveEnhancedKeyUsageIdentifier

The CertRemoveEnhancedKeyUsageIdentifier function removes a usage identifier object identifier (OID) from the enhanced key usage (EKU) extended property of the certificate.

CertRemoveStoreFromCollection

Removes a sibling certificate store from a collection store.

CertResyncCertificateChainEngine

Resyncs the certificate chain engine, which resynchronizes the stores the store's engine and updates the engine caches.

CertRetrieveLogoOrBiometricInfo

Performs a URL retrieval of logo or biometric information specified in either the szOID_LOGOTYPE_EXT or szOID_BIOMETRIC_EXT certificate extension.

CertSaveStore

Saves the certificate store to a file or to a memory BLOB.

CertSelectCertificateChains

Retrieves certificate chains based on specified selection criteria.

CertSerializeCertificateStoreElement

The CertSerializeCertificateStoreElement function serializes a certificate context's encoded certificate and its encoded properties. The result can be persisted to storage so that the certificate and properties can be retrieved at a later time.

CertSerializeCRLStoreElement

The CertSerializeCRLStoreElement function serializes an encoded certificate revocation list (CRL) context and the encoded representation of its properties.

CertSerializeCTLStoreElement

The CertSerializeCTLStoreElement function serializes an encoded certificate trust list (CTL) context and the encoded representation of its properties. The result can be persisted to storage so that the CTL and properties can be retrieved later.

CertSetCertificateContextPropertiesFromCTLEntry

Sets the properties on the certificate context by using the attributes in the specified certificate trust list (CTL) entry.

CertSetCertificateContextProperty

Sets an extended property for a specified certificate context.

CertSetCRLContextProperty

Sets an extended property for the specified certificate revocation list (CRL) context.

CertSetCTLContextProperty

Sets an extended property for the specified certificate trust list (CTL) context.

CertSetEnhancedKeyUsage

The CertSetEnhancedKeyUsage function sets the enhanced key usage (EKU) property for the certificate.

CertSetStoreProperty

The CertSetStoreProperty function sets a store property.

CertStrToNameA

Converts a null-terminated X.500 string to an encoded certificate name.

CertStrToNameW

Converts a null-terminated X.500 string to an encoded certificate name.

CertUnregisterPhysicalStore

The CertUnregisterPhysicalStore function removes a physical store from a specified system store collection. CertUnregisterPhysicalStore can also be used to delete the physical store.

CertUnregisterSystemStore

The CertUnregisterSystemStore function unregisters a specified system store.

CertVerifyCertificateChainPolicy

Checks a certificate chain to verify its validity, including its compliance with any specified validity policy criteria.

CertVerifyCRLRevocation

Check a certificate revocation list (CRL) to determine whether a subject's certificate has or has not been revoked.

CertVerifyCRLTimeValidity

The CertVerifyCRLTimeValidity function verifies the time validity of a CRL.

CertVerifyCTLUsage

Verifies that a subject is trusted for a specified usage by finding a signed and time-valid certificate trust list (CTL) with the usage identifiers that contain the subject.

CertVerifyRevocation

Checks the revocation status of the certificates contained in the rgpvContext array. If a certificate in the list is found to be revoked, no further checking is done.

CertVerifySubjectCertificateContext

The CertVerifySubjectCertificateContext function performs the enabled verification checks on a certificate by checking the validity of the certificate's issuer. The new Certificate Chain Verification Functions are recommended instead of this function.

CertVerifyTimeValidity

The CertVerifyTimeValidity function verifies the time validity of a certificate.

CertVerifyValidityNesting

The CertVerifyValidityNesting function verifies that a subject certificate's time validity nests correctly within its issuer's time validity.

CryptAcquireCertificatePrivateKey

Obtains the private key for a certificate.

CryptAcquireContextA

Used to acquire a handle to a particular key container within a particular cryptographic service provider (CSP). This returned handle is used in calls to CryptoAPI functions that use the selected CSP.

CryptAcquireContextW

Used to acquire a handle to a particular key container within a particular cryptographic service provider (CSP). This returned handle is used in calls to CryptoAPI functions that use the selected CSP.

CryptBinaryToStringA

Converts an array of bytes into a formatted string.

CryptBinaryToStringW

Converts an array of bytes into a formatted string.

CryptCloseAsyncHandle

CryptContextAddRef

Adds one to the reference count of an HCRYPTPROV cryptographic service provider (CSP) handle.

CryptCreateAsyncHandle

CryptCreateHash

Initiates the hashing of a stream of data. It creates and returns to the calling application a handle to a cryptographic service provider (CSP) hash object.

CryptCreateKeyIdentifierFromCSP

Important This API is deprecated.

CryptDecodeMessage

Decodes, decrypts, and verifies a cryptographic message.

CryptDecodeObject

The CryptDecodeObject function decodes a structure of the type indicated by the lpszStructType parameter. The use of CryptDecodeObjectEx is recommended as an API that performs the same function with significant performance improvements.

CryptDecodeObjectEx

Decodes a structure of the type indicated by the lpszStructType parameter.

CryptDecrypt

Decrypts data previously encrypted by using the CryptEncrypt function.

CryptDecryptAndVerifyMessageSignature

The CryptDecryptAndVerifyMessageSignature function decrypts a message and verifies its signature.

CryptDecryptMessage

The CryptDecryptMessage function decodes and decrypts a message.

CryptDeriveKey

Generates cryptographic session keys derived from a base data value.

CryptDestroyHash

Destroys the hash object referenced by the hHash parameter.

CryptDestroyKey

Releases the handle referenced by the hKey parameter.

CryptDuplicateHash

Makes an exact copy of a hash to the point when the duplication is done.

CryptDuplicateKey

Makes an exact copy of a key and the state of the key.

CryptEncodeObject

The CryptEncodeObject function encodes a structure of the type indicated by the value of the lpszStructType parameter. The use of CryptEncodeObjectEx is recommended as an API that performs the same function with significant performance improvements.

CryptEncodeObjectEx

Encodes a structure of the type indicated by the value of the lpszStructType parameter.

CryptEncrypt

Encrypts data. The algorithm used to encrypt the data is designated by the key held by the CSP module and is referenced by the hKey parameter.

CryptEncryptMessage

The CryptEncryptMessage function encrypts and encodes a message.

CryptEnumKeyIdentifierProperties

The CryptEnumKeyIdentifierProperties function enumerates key identifiers and their properties.

CryptEnumOIDFunction

The CryptEnumOIDFunction function enumerates the registered object identifier (OID) functions.

CryptEnumOIDInfo

Enumerates predefined and registered object identifier (OID) CRYPT_OID_INFO structures. This function enumerates either all of the predefined and registered structures or only structures identified by a selected OID group.

CryptEnumProvidersA

Important This API is deprecated.

CryptEnumProvidersW

Important This API is deprecated.

CryptEnumProviderTypesA

Retrieves the first or next types of cryptographic service provider (CSP) supported on the computer.

CryptEnumProviderTypesW

Retrieves the first or next types of cryptographic service provider (CSP) supported on the computer.

CryptExportKey

Exports a cryptographic key or a key pair from a cryptographic service provider (CSP) in a secure manner.

CryptExportPKCS8

Exports the private key in PKCS

CryptExportPKCS8Ex

Exports the private key in PKCS

CryptExportPublicKeyInfo

The CryptExportPublicKeyInfo function exports the public key information associated with the corresponding private key of the provider. For an updated version of this function, see CryptExportPublicKeyInfoEx.

CryptExportPublicKeyInfoEx

Exports the public key information associated with the provider's corresponding private key.

CryptExportPublicKeyInfoFromBCryptKeyHandle

Exports the public key information associated with a provider's corresponding private key.

CryptFindCertificateKeyProvInfo

Enumerates the cryptographic providers and their containers to find the private key that corresponds to the certificate's public key.

CryptFindLocalizedName

Finds the localized name for the specified name, such as the localize name of the "Root" system store.

CryptFindOIDInfo

Retrieves the first predefined or registered CRYPT_OID_INFO structure that matches a specified key type and key. The search can be limited to object identifiers (OIDs) within a specified OID group.

CryptFormatObject

The CryptFormatObject function formats the encoded data and returns a Unicode string in the allocated buffer according to the certificate encoding type.

CryptFreeOIDFunctionAddress

The CryptFreeOIDFunctionAddress function releases a handle returned by CryptGetOIDFunctionAddress or CryptGetDefaultOIDFunctionAddress by decrementing the reference count on the function handle.

CryptGenKey

Generates a random cryptographic session key or a public/private key pair. A handle to the key or key pair is returned in phKey. This handle can then be used as needed with any CryptoAPI function that requires a key handle.

CryptGenRandom

Fills a buffer with cryptographically random bytes.

CryptGetAsyncParam

CryptGetDefaultOIDDllList

The CryptGetDefaultOIDDllList function acquires the list of the names of DLL files that contain registered default object identifier (OID) functions for a specified function set and encoding type.

CryptGetDefaultOIDFunctionAddress

The CryptGetDefaultOIDFunctionAddress function loads the DLL that contains a default function address.

CryptGetDefaultProviderA

Finds the default cryptographic service provider (CSP) of a specified provider type for the local computer or current user.

CryptGetDefaultProviderW

Finds the default cryptographic service provider (CSP) of a specified provider type for the local computer or current user.

CryptGetHashParam

Retrieves data that governs the operations of a hash object.

CryptGetKeyIdentifierProperty

The CryptGetKeyIdentifierProperty acquires a specific property from a specified key identifier.

CryptGetKeyParam

Retrieves data that governs the operations of a key.

CryptGetMessageCertificates

The CryptGetMessageCertificates function returns the handle of an open certificate store containing the message's certificates and CRLs. This function calls CertOpenStore using provider

type CERT_STORE_PROV_PKCS7 as its lpszStoreProvider parameter.

CryptGetMessageSignerCount

The CryptGetMessageSignerCount function returns the number of signers of a signed message.

CryptGetObjectUrl

Acquires the URL of the remote object from a certificate, certificate trust list (CTL), or certificate revocation list (CRL).

CryptGetOIDFunctionAddress

Searches the list of registered and installed functions for an encoding type and object identifier (OID) match.

CryptGetOIDFunctionValue

The CryptGetOIDFunctionValue function queries a value associated with an OID.

CryptGetProvParam

Retrieves parameters that govern the operations of a cryptographic service provider (CSP).

CryptGetTimeValidObject

Retrieves a CRL, an OCSP response, or CTL object that is valid within a given context and time.

CryptGetUserKey

Retrieves a handle of one of a user's two public/private key pairs.

CryptHashCertificate

The CryptHashCertificate function hashes the entire encoded content of a certificate including its signature.

CryptHashCertificate2

Hashes a block of data by using a CNG hash provider.

CryptHashData

Adds data to a specified hash object.

CryptHashMessage

Creates a hash of the message.

CryptHashPublicKeyInfo

Encodes the public key information in a CERT_PUBLIC_KEY_INFO structure and computes the hash of the encoded bytes.

CryptHashSessionKey

Computes the cryptographic hash of a session key object.

CryptHashToBeSigned

Important This API is deprecated.

CryptImportKey

Transfers a cryptographic key from a key BLOB into a cryptographic service provider (CSP).

CryptImportPKCS8

Imports the private key in PKCS

CryptImportPublicKeyInfo

Converts and imports the public key information into the provider and returns a handle of the public key.

CryptImportPublicKeyInfoEx

Important This API is deprecated.

CryptImportPublicKeyInfoEx2

Imports a public key into the CNG asymmetric provider that corresponds to the public key object identifier (OID) and returns a CNG handle to the key.

CryptInitOIDFunctionSet

The CryptInitOIDFunctionSet initializes and returns the handle of the OID function set identified by a supplied function set name.

CryptInstallDefaultContext

Installs a specific provider to be the default context provider for the specified algorithm.

CryptInstallOIDFunctionAddress

The CryptInstallOIDFunctionAddress function installs a set of callable object identifier (OID) function addresses.

CryptMemAlloc

The CryptMemAlloc function allocates memory for a buffer. It is used by all Crypt32.lib functions that return allocated buffers.

CryptMemFree

The CryptMemFree function frees memory allocated by CryptMemAlloc or CryptMemRealloc.

CryptMemRealloc

The CryptMemRealloc function frees the memory currently allocated for a buffer and allocates memory for a new buffer.

CryptMsgCalculateEncodedLength

Calculates the maximum number of bytes needed for an encoded cryptographic message given

the message type, encoding parameters, and total length of the data to be encoded.

CryptMsgClose

The CryptMsgClose function closes a cryptographic message handle. At each call to this function, the reference count on the message is reduced by one. When the reference count reaches zero, the message is fully released.

CryptMsgControl

Performs a control operation after a message has been decoded by a final call to the CryptMsgUpdate function.

CryptMsgCountersign

Countersigns an existing signature in a message.

CryptMsgCountersignEncoded

Countersigns an existing PKCS

CryptMsgDuplicate

The CryptMsgDuplicate function duplicates a cryptographic message handle by incrementing its reference count.

CryptMsgEncodeAndSignCTL

The CryptMsgEncodeAndSignCTL function encodes a CTL and creates a signed message containing the encoded CTL.This function first encodes the CTL pointed to by pCtlInfo and then calls CryptMsgSignCTL to sign the encoded message.

CryptMsgGetAndVerifySigner

The CryptMsgGetAndVerifySigner function verifies a cryptographic message's signature.

CryptMsgGetParam

Acquires a message parameter after a cryptographic message has been encoded or decoded.

CryptMsgOpenToDecode

Opens a cryptographic message for decoding and returns a handle of the opened message.

CryptMsgOpenToEncode

Opens a cryptographic message for encoding and returns a handle of the opened message.

CryptMsgSignCTL

The CryptMsgSignCTL function creates a signed message containing an encoded CTL.

CryptMsgUpdate

Adds contents to a cryptographic message.

CryptMsgVerifyCountersignatureEncoded

Verifies a countersignature in terms of the SignerInfo structure (as defined by PKCS

CryptMsgVerifyCountersignatureEncodedEx

Verifies that the pbSignerInfoCounterSignature parameter contains the encrypted hash of the encryptedDigest field of the pbSignerInfo parameter structure.

CryptQueryObject

Retrieves information about the contents of a cryptography API object, such as a certificate, a certificate revocation list, or a certificate trust list.

CryptRegisterDefaultOIDFunction

The CryptRegisterDefaultOIDFunction registers a DLL containing the default function to be called for the specified encoding type and function name. Unlike CryptRegisterOIDFunction, the function name to be exported by the DLL cannot be overridden.

CryptRegisterOIDFunction

Registers a DLL that contains the function to be called for the specified encoding type, function name, and object identifier (OID).

CryptRegisterOIDInfo

The CryptRegisterOIDInfo function registers the OID information specified in the CRYPT_OID_INFO structure, persisting it to the registry.

CryptReleaseContext

Releases the handle of a cryptographic service provider (CSP) and a key container.

CryptRetrieveObjectByUrlA

Retrieves the public key infrastructure (PKI) object from a location specified by a URL.

CryptRetrieveObjectByUrlW

Retrieves the public key infrastructure (PKI) object from a location specified by a URL.

CryptRetrieveTimeStamp

Encodes a time stamp request and retrieves the time stamp token from a location specified by a URL to a Time Stamping Authority (TSA).

CryptSetAsyncParam

CryptSetHashParam

Customizes the operations of a hash object, including setting up initial hash contents and selecting a specific hashing algorithm.

CryptSetKeyIdentifierProperty

The CryptSetKeyIdentifierProperty function sets the property of a specified key identifier. This function can set the property on the computer identified in pwszComputerName.

CryptSetKeyParam

Customizes various aspects of a session key's operations.

CryptSetOIDFunctionValue

The CryptSetOIDFunctionValue function sets a value for the specified encoding type, function name, OID, and value name.

CryptSetProviderA

Specifies the current user's default cryptographic service provider (CSP).

CryptSetProviderExA

Specifies the default cryptographic service provider (CSP) of a specified provider type for the local computer or current user.

CryptSetProviderExW

Specifies the default cryptographic service provider (CSP) of a specified provider type for the local computer or current user.

CryptSetProviderW

Specifies the current user's default cryptographic service provider (CSP).

CryptSetProvParam

Customizes the operations of a cryptographic service provider (CSP). This function is commonly used to set a security descriptor on the key container associated with a CSP to control access to the private keys in that key container.

CryptSignAndEncodeCertificate

Encodes and signs a certificate, certificate revocation list (CRL), certificate trust list (CTL), or certificate request.

CryptSignAndEncryptMessage

The CryptSignAndEncryptMessage function creates a hash of the specified content, signs the hash, encrypts the content, hashes the encrypted contents and the signed hash, and then encodes both the encrypted content and the signed hash.

CryptSignCertificate

The CryptSignCertificate function signs the "to be signed" information in the encoded signed content.

CryptSignHashA

Signs data.

CryptSignHashW

Signs data.

CryptSignMessage

The CryptSignMessage function creates a hash of the specified content, signs the hash, and then encodes both the original message content and the signed hash.

CryptSignMessageWithKey

Signs a message by using a CSP's private key specified in the parameters.

CryptStringToBinaryA

Converts a formatted string into an array of bytes.

CryptStringToBinaryW

Converts a formatted string into an array of bytes.

CryptUninstallDefaultContext

Important This API is deprecated.

CryptUnregisterDefaultOIDFunction

The CryptUnregisterDefaultOIDFunction removes the registration of a DLL containing the default function to be called for the specified encoding type and function name.

CryptUnregisterOIDFunction

Removes the registration of a DLL that contains the function to be called for the specified encoding type, function name, and OID.

CryptUnregisterOIDInfo

The CryptUnregisterOIDInfo function removes the registration of a specified CRYPT_OID_INFO OID information structure. The structure to be unregistered is identified by the structure's pszOID and dwGroupId members.

CryptVerifyCertificateSignature

Verifies the signature of a certificate, certificate revocation list (CRL), or certificate request by using the public key in a CERT_PUBLIC_KEY_INFO structure.

CryptVerifyCertificateSignatureEx

Verifies the signature of a subject certificate, certificate revocation list, certificate request, or keygen request by using the issuer's public key.

CryptVerifyDetachedMessageHash

The CryptVerifyDetachedMessageHash function verifies a detached hash.

CryptVerifyDetachedMessageSignature

The CryptVerifyDetachedMessageSignature function verifies a signed message containing a detached signature or signatures.

CryptVerifyMessageHash

The CryptVerifyMessageHash function verifies the hash of specified content.

CryptVerifyMessageSignature

Verifies a signed message's signature.

CryptVerifyMessageSignatureWithKey

Verifies a signed message's signature by using specified public key information.

CryptVerifySignatureA

Verifies the signature of a hash object.

CryptVerifySignatureW

Verifies the signature of a hash object.

CryptVerifyTimeStampSignature

Validates the time stamp signature on a specified array of bytes.

GetEncSChannel

This function is unavailable.

PFXExportCertStore

Exports the certificates and, if available, the associated private keys from the referenced certificate store.

PFXExportCertStoreEx

Exports the certificates and, if available, their associated private keys from the referenced certificate store.

PFXImportCertStore

Imports a PFX BLOB and returns the handle of a store that contains certificates and any associated private keys.

PFXIsPFXBlob

The PFXIsPFXBlob function attempts to decode the outer layer of a BLOB as a PFX packet.

PFXVerifyPassword

The PFXVerifyPassword function attempts to decode the outer layer of a BLOB as a Personal Information Exchange (PFX) packet and to decrypt it with the given password. No data from the BLOB is imported.

Callback functions

PCRYPT_DECRYPT_PRIVATE_KEY_FUNC

Decrypts the private key and returns the decrypted key in the pbClearTextKey parameter.

PCRYPT_ENCRYPT_PRIVATE_KEY_FUNC

Encrypts the private key and returns the encrypted contents in the pbEncryptedKey parameter.

PCRYPT_RESOLVE_HCRYPTPROV_FUNC

Returns a handle to a cryptographic service provider (CSP) by using the phCryptProv parameter to receive the key being imported.

PFN_CERT_CHAIN_FIND_BY_ISSUER_CALLBACK

An application-defined callback function that allows the application to filter certificates that might be added to the certificate chain.

PFN_CERT_CREATE_CONTEXT_SORT_FUNC

Called for each sorted context entry when a context is created.

PFN_CERT_DLL_OPEN_STORE_PROV_FUNC

Implemented by a store-provider and is used to open a store.

PFN_CERT_ENUM_PHYSICAL_STORE

The CertEnumPhysicalStoreCallback callback function formats and presents information on each physical store found by a call to CertEnumPhysicalStore.

PFN_CERT_ENUM_SYSTEM_STORE

The CertEnumSystemStoreCallback callback function formats and presents information on each system store found by a call to CertEnumSystemStore.

PFN_CERT_ENUM_SYSTEM_STORE_LOCATION

The CertEnumSystemStoreLocationCallback callback function formats and presents information on each system store location found by a call to CertEnumSystemStoreLocation.

PFN_CERT_STORE_PROV_CLOSE

An application-defined callback function that is called by CertCloseStore when the store's reference count is decremented to zero.

PFN_CERT_STORE_PROV_CONTROL

The CertStoreProvControl callback function supports the CertControlStore API. All of the API's parameters are passed straight through to the callback. For details, see CertControlStore.

PFN_CERT_STORE_PROV_DELETE_CERT

An application-defined callback function that is called by CertDeleteCertificateFromStore before deleting a certificate from the store.

PFN_CERT_STORE_PROV_DELETE_CRL

An application-defined callback function that is called by CertDeleteCRLFromStore before deleting the CRL from the store.

PFN_CERT_STORE_PROV_READ_CERT

An application-defined callback function that reads the provider's copy of the certificate context.

PFN_CERT_STORE_PROV_READ_CRL

An application-defined callback function that reads the provider's copy of the CRL context.

PFN_CERT_STORE_PROV_READ_CTL

The CertStoreProvReadCTL callback function is called to read the provider's copy of the CTL context and, if it exists, to create a new CTL context.

PFN_CERT_STORE_PROV_SET_CERT_PROPERTY

An application-defined callback function that is called by CertSetCertificateContextProperty before setting the certificate's property.

PFN_CERT_STORE_PROV_SET_CRL_PROPERTY

An application-defined callback function that is called by CertSetCRLContextProperty before setting the CRL's property.

PFN_CERT_STORE_PROV_SET_CTL_PROPERTY

The CertStoreProvSetCTLProperty callback function determines whether a property can be set on a CTL.

PFN_CERT_STORE_PROV_WRITE_CERT

An application-defined callback function that is called by CertAddEncodedCertificateToStore, CertAddCertificateContextToStore and CertAddSerializedElementToStore before adding to the store.

PFN_CERT_STORE_PROV_WRITE_CRL

An application-defined callback function that is called by CertAddEncodedCRLToStore, CertAddCRLContextToStore and CertAddSerializedElementToStore before adding to the store.

PFN_CERT_STORE_PROV_WRITE_CTL

The CertStoreProvWriteCTL callback function can be called by CertAddEncodedCTLToStore, CertAddCTLContextToStore or CertAddSerializedElementToStore before a CTL is added to the store.

PFN_CMSG_CNG_IMPORT_CONTENT_ENCRYPT_KEY

Imports an already decrypted content encryption key (CEK).

PFN_CMSG_CNG_IMPORT_KEY_AGREE

Decrypts a content encryption key (CEK) that is intended for a key agreement recipient.

PFN_CMSG_CNG_IMPORT_KEY_TRANS

Imports and decrypts a content encryption key (CEK) that is intended for a key transport recipient.

PFN_CMSG_EXPORT_KEY_AGREE

Encrypts and exports the content encryption key for a key agreement recipient of an enveloped message.

PFN_CMSG_EXPORT_KEY_TRANS

Encrypts and exports the content encryption key for a key transport recipient of an enveloped message.

PFN_CMSG_EXPORT_MAIL_LIST

Encrypts and exports the content encryption key for a mailing list recipient of an enveloped message.

PFN_CMSG_GEN_CONTENT_ENCRYPT_KEY

Generates the symmetric key used to encrypt content for an enveloped message.

PFN_CMSG_IMPORT_KEY_AGREE

Imports a content encryption key for a key transport recipient of an enveloped message.

PFN_CMSG_IMPORT_KEY_TRANS

Imports a content encryption key for a key transport recipient of an enveloped message.

PFN_CMSG_IMPORT_MAIL_LIST

Imports a content encryption key for a key transport recipient of an enveloped message.

PFN_CRYPT_ENUM_KEYID_PROP

The CRYPT_ENUM_KEYID_PROP callback function is used with the CryptEnumKeyIdentifierProperties function.

PFN_CRYPT_ENUM_OID_FUNC

The CRYPT_ENUM_OID_FUNCTION callback function is used with the CryptEnumOIDFunction function.

PFN_CRYPT_ENUM_OID_INFO

The CRYPT_ENUM_OID_INFO callback function is used with the CryptEnumOIDInfo function.

PFN_CRYPT_EXPORT_PUBLIC_KEY_INFO_EX2_FUNC

Called by CryptExportPublicKeyInfoEx to export a public key BLOB and encode it.

PFN_CRYPT_EXTRACT_ENCODED_SIGNATURE_PARAMETERS_FUNC

Called to decode and return the hash algorithm identifier and optionally the signature parameters.

PFN_CRYPT_GET_SIGNER_CERTIFICATE

The CryptGetSignerCertificateCallback user supplied callback function is used with the CRYPT_VERIFY_MESSAGE_PARA structure to get and verify a message signer's certificate.

PFN_CRYPT_OBJECT_LOCATOR_PROVIDER_FLUSH

Specifies that an object has changed.

PFN_CRYPT_OBJECT_LOCATOR_PROVIDER_FREE

Releases the object returned by the provider.

PFN_CRYPT_OBJECT_LOCATOR_PROVIDER_FREE_IDENTIFIER

Releases memory for an object identifier.

PFN_CRYPT_OBJECT_LOCATOR_PROVIDER_FREE_PASSWORD

Releases the password used to encrypt a personal information exchange (PFX) byte array.

PFN_CRYPT_OBJECT_LOCATOR_PROVIDER_GET

Retrieves an object.

PFN_CRYPT_OBJECT_LOCATOR_PROVIDER_INITIALIZE

Initializes the provider.

PFN_CRYPT_OBJECT_LOCATOR_PROVIDER_RELEASE

Releases the provider.

PFN_CRYPT_SIGN_AND_ENCODE_HASH_FUNC

Called to sign and encode a computed hash.

PFN_CRYPT_VERIFY_ENCODED_SIGNATURE_FUNC

Called to decrypt an encoded signature and compare it to a computed hash.

PFN_IMPORT_PUBLIC_KEY_INFO_EX2_FUNC

Called by CryptImportPublicKeyInfoEx2 to decode the public key algorithm identifier, load the

algorithm provider, and import the key pair.

Structures

AUTHENTICODE_EXTRA_CERT_CHAIN_POLICY_PARA

Holds policy information used in the verification of certificate chains for files.

AUTHENTICODE_EXTRA_CERT_CHAIN_POLICY_STATUS

The AUTHENTICODE_EXTRA_CERT_CHAIN_POLICY_STATUS structure holds additional Authenticode policy information for chain verification of files.

AUTHENTICODE_TS_EXTRA_CERT_CHAIN_POLICY_PARA

The AUTHENTICODE_TS_EXTRA_CERT_CHAIN_POLICY_PARA structure contains time stamp policy information that can be used in certificate chain verification of files.

BLOBHEADER

Indicates a key's BLOB type and the algorithm that the key uses.

CERT_ACCESS_DESCRIPTION

The CERT_ACCESS_DESCRIPTION structure is a member of a CERT_AUTHORITY_INFO_ACCESS structure.

CERT_ALT_NAME_ENTRY

Contains an alternative name in one of a variety of name forms.

CERT_ALT_NAME_INFO

The CERT_ALT_NAME_INFO structure is used in encoding and decoding extensions for subject or issuer certificates, Certificate Revocation Lists (CRLs), and Certificate Trust Lists (CTLs).

CERT_AUTHORITY_INFO_ACCESS

Represents authority information access and subject information access certificate extensions and specifies how to access additional information and services for the subject or the issuer of a certificate.

CERT_AUTHORITY_KEY_ID_INFO

Identifies the key used to sign a certificate or certificate revocation list (CRL).

CERT_AUTHORITY_KEY_ID2_INFO

The CERT_AUTHORITY_KEY_ID2_INFO structure identifies the key used to sign a certificate or CRL.

CERT_BASIC_CONSTRAINTS_INFO

The CERT_BASIC_CONSTRAINTS_INFO structure contains information that indicates whether the certified subject can act as a certification authority (CA), an end entity, or both.

CERT_BASIC_CONSTRAINTS2_INFO

The CERT_BASIC_CONSTRAINTS2_INFO structure contains information indicating whether the certified subject can act as a CA or an end entity. If the subject can act as a CA, a certification path length constraint can also be specified.

CERT_BIOMETRIC_DATA

Contains information about biometric data.

CERT_BIOMETRIC_EXT_INFO

Contains a set of biometric information.

CERT_CHAIN_CONTEXT

Contains an array of simple certificate chains and a trust status structure that indicates summary validity data on all of the connected simple chains.

CERT_CHAIN_ELEMENT

The CERT_CHAIN_ELEMENT structure is a single element in a simple certificate chain.

CERT_CHAIN_ENGINE_CONFIG

Sets parameters for building a non-default certificate chain engine. The engine used determines the ways that certificate chains are built.

CERT_CHAIN_FIND_ISSUER_PARA

Contains information used in the CertFindChainInStore function to build certificate chains.

CERT_CHAIN_PARA

The CERT_CHAIN_PARA structure establishes the searching and matching criteria to be used in building a certificate chain.

CERT_CHAIN_POLICY_PARA

Contains information used in CertVerifyCertificateChainPolicy to establish policy criteria for the verification of certificate chains.

CERT_CHAIN_POLICY_STATUS

Holds certificate chain status information returned by the CertVerifyCertificateChainPolicy function when the certificate chains are validated.

CERT_CONTEXT

Contains both the encoded and decoded representations of a certificate.

CERT_CREATE_CONTEXT_PARA

Defines additional values that can be used when calling the CertCreateContext function.

CERT_CRL_CONTEXT_PAIR

The CERT_CRL_CONTEXT_PAIR structure contains a certificate context and an associated CRL context.

CERT_DH_PARAMETERS

Contains parameters associated with a Diffie/Hellman public key algorithm.

CERT_DSS_PARAMETERS

Contains parameters associated with a Digital Signature Standard (DSS) public key algorithm.

CERT_ECC_SIGNATURE

Contains the r and s values for an Elliptic Curve Digital Signature Algorithm (ECDSA) signature.

CERT_EXTENSION

The CERT_EXTENSION structure contains the extension information for a certificate, Certificate Revocation List (CRL) or Certificate Trust List (CTL).

CERT_EXTENSIONS

The CERT_EXTENSIONS structure contains an array of extensions.

CERT_GENERAL_SUBTREE

The CERT_GENERAL_SUBTREE structure is used in CERT_NAME_CONSTRAINTS_INFO structure. This structure provides the identity of a certificate that can be included or excluded.

CERT_HASHED_URL

Contains a hashed URL.

CERT_ID

Is used as a flexible means of uniquely identifying a certificate.

CERT_INFO

Contains the information of a certificate.

CERT_ISSUER_SERIAL_NUMBER

Acts as a unique identifier of a certificate containing the issuer and issuer's serial number for a certificate.

CERT_KEY_ATTRIBUTES_INFO

The CERT_KEY_ATTRIBUTES_INFO structure contains optional additional information about the public key being certified.

CERT_KEY_CONTEXT

Contains data associated with a CERT_KEY_CONTEXT_PROP_ID property.

CERT_KEY_USAGE_RESTRICTION_INFO

The CERT_KEY_USAGE_RESTRICTION_INFO structure contains restrictions imposed on the usage of a certificate's public key. This includes purposes for use of the key and policies under which the key can be used.

CERT_KEYGEN_REQUEST_INFO

Contains information stored in the Netscape key generation request. The subject and subject public key BLOBs are encoded.

CERT_LDAP_STORE_OPENED_PARA

Used with the CertOpenStore function when the CERT_STORE_PROV_LDAP provider is specified by using the CERT_LDAP_STORE_OPENED_FLAG flag to specify both the existing LDAP session to use to perform the query as well as the LDAP query string.

CERT_LOGOTYPE_AUDIO

Contains information about an audio logotype.

CERT_LOGOTYPE_AUDIO_INFO

Contains more detailed information about an audio logotype.

CERT_LOGOTYPE_DATA

Contains logotype data.

CERT_LOGOTYPE_DETAILS

Contains additional information about a logotype.

CERT_LOGOTYPE_EXT_INFO

Contains a set of logotype information.

CERT_LOGOTYPE_IMAGE

Contains information about an image logotype.

CERT_LOGOTYPE_IMAGE_INFO

Contains more detailed information about an image logotype.

CERT_LOGOTYPE_INFO

Contains information about logotype data.

CERT_LOGOTYPE_REFERENCE

Contains logotype reference information.

CERT_NAME_CONSTRAINTS_INFO

The CERT_NAME_CONSTRAINTS_INFO structure contains information about certificates that are specifically permitted or excluded from trust.

CERT_NAME_INFO

Contains subject or issuer names.

CERT_NAME_VALUE

Contains a relative distinguished name (RDN) attribute value.

CERT_OR_CRL_BLOB

Encapsulates certificates for use with Internet Key Exchange messages.

CERT_OR_CRL_BUNDLE

Encapsulates an array of certificates for use with Internet Key Exchange messages.

CERT_OTHER_LOGOTYPE_INFO

Contains information about logo types that are not predefined.

CERT_PAIR

The CERT_PAIR structure contains a certificate and its pair cross certificate.

CERT_PHYSICAL_STORE_INFO

Contains information on physical certificate stores.

CERT_POLICIES_INFO

The CERT_POLICIES_INFO structure contains an array of CERT_POLICY_INFO.

CERT_POLICY_CONSTRAINTS_INFO

The CERT_POLICY_CONSTRAINTS_INFO structure contains established policies for accepting certificates as trusted.

CERT_POLICY_ID

The CERT_POLICY_ID structure contains a list of certificate policies that the certificate expressly supports, together with optional qualifier information pertaining to these policies.

CERT_POLICY_INFO

The CERT_POLICY_INFO structure contains an object identifier (OID) specifying a policy and an optional array of policy qualifiers.

CERT_POLICY_MAPPING

Contains a mapping between issuer domain and subject domain policy OIDs.

CERT_POLICY_MAPPINGS_INFO

The CERT_POLICY_MAPPINGS_INFO structure provides mapping between the policy OIDs of two domains.

CERT_POLICY_QUALIFIER_INFO

The CERT_POLICY_QUALIFIER_INFO structure contains an object identifier (OID) specifying the qualifier and qualifier-specific supplemental information.

CERT_PRIVATE_KEY_VALIDITY

The CERT_PRIVATE_KEY_VALIDITY structure indicates a valid time span for the private key corresponding to a certificate's public key.

CERT_PUBLIC_KEY_INFO

Contains a public key and its algorithm.

CERT_QC_STATEMENT

Represents a single statement in a sequence of one or more statements for inclusion in a Qualified Certificate (QC) statements extension.

CERT_QC_STATEMENTS_EXT_INFO

Contains a sequence of one or more statements that make up the Qualified Certificate (QC) statements extension for a QC.

CERT_RDN

The CERT_RDN structure contains a relative distinguished name (RDN) consisting of an array of CERT_RDN_ATTR structures.

CERT_RDN_ATTR

Contains a single attribute of a relative distinguished name (RDN). A whole RDN is expressed in a CERT_RDN structure that contains an array of CERT_RDN_ATTR structures.

CERT_REQUEST_INFO

The CERT_REQUEST_INFO structure contains information for a certificate request. The subject, subject public key, and attribute BLOBs are encoded.

CERT_REVOCATION_CHAIN_PARA

Contains parameters used for building a chain for an independent online certificate status protocol (OCSP) response signer certificate.

CERT_REVOCATION_CRL_INFO

Contains information updated by a certificate revocation list (CRL) revocation type handler.

CERT_REVOCATION_INFO

Indicates the revocation status of a certificate in a CERT_CHAIN_ELEMENT.

CERT_REVOCATION_PARA

Is passed in calls to the CertVerifyRevocation function to assist in finding the issuer of the context to be verified.

CERT_REVOCATION_STATUS

Contains information on the revocation status of the certificate.

CERT_SELECT_CHAIN_PARA

Contains the parameters used for building and selecting chains.

CERT_SELECT_CRITERIA

Specifies selection criteria that is passed to the CertSelectCertificateChains function.

CERT_SERVER_OCSP_RESPONSE_CONTEXT

Contains an encoded OCSP response.

CERT_SIGNED_CONTENT_INFO

The CERT_SIGNED_CONTENT_INFO structure contains encoded content to be signed and a BLOB to hold the signature. The ToBeSigned member is an encoded CERT_INFO, CRL_INFO, CTL_INFO or CERT_REQUEST_INFO.

CERT_SIMPLE_CHAIN

The CERT_SIMPLE_CHAIN structure contains an array of chain elements and a summary trust status for the chain that the array represents.

CERT_STORE_PROV_FIND_INFO

Used by many of the store provider callback functions.

CERT_STORE_PROV_INFO

Contains information returned by the installed CertDllOpenStoreProv function when a store is opened by using the CertOpenStore function.

CERT_STRONG_SIGN_PARA

Contains parameters used to check for strong signatures on certificates, certificate revocation lists (CRLs), online certificate status protocol (OCSP) responses, and PKCS

CERT_STRONG_SIGN_SERIALIZED_INFO

Contains the signature algorithm/hash algorithm and public key algorithm/bit length pairs that

can be used for strong signing.

CERT_SYSTEM_STORE_INFO

The CERT_SYSTEM_STORE_INFO structure contains information used by functions that work with system stores. Currently, no essential information is contained in this structure.

CERT_SYSTEM_STORE_RELOCATE_PARA

The CERT_SYSTEM_STORE_RELOCATE_PARA structure contains data to be passed to CertOpenStore when that function's dwFlags parameter is set to CERT_SYSTEM_STORE_RELOCATE_FLAG.

CERT_TEMPLATE_EXT

A certificate template.

CERT_TRUST_LIST_INFO

The CERT_TRUST_LIST_INFO structure that indicates valid usage of a CTL.

CERT_TRUST_STATUS

Contains trust information about a certificate in a certificate chain, summary trust information about a simple chain of certificates, or summary information about an array of simple chains.

CERT_USAGE_MATCH

Provides criteria for identifying issuer certificates to be used to build a certificate chain.

CERT_X942_DH_PARAMETERS

Contains parameters associated with a Diffie-Hellman public key algorithm.

CERT_X942_DH_VALIDATION_PARAMS

Optionally pointed to by a member of the CERT_X942_DH_PARAMETERS structure and contains additional seed information.

CMC_ADD_ATTRIBUTES_INFO

Contains certificate attributes to be added to a certificate.

CMC_ADD_EXTENSIONS_INFO

Contains certificate extension control attributes to be added to a certificate.

CMC_DATA_INFO

Provides a means of communicating different pieces of tagged information.

CMC_PEND_INFO

A possible member of a CMC_STATUS_INFO structure.

CMC_RESPONSE_INFO

Provides a means of communicating different pieces of tagged information.

CMC_STATUS_INFO

Contains status information about Certificate Management Messages over CMS.

CMC_TAGGED_ATTRIBUTE

Used in the CMC_DATA_INFO and CMC_RESPONSE_INFO structures.

CMC_TAGGED_CERT_REQUEST

Used in the CMC_TAGGED_REQUEST structure.

CMC_TAGGED_CONTENT_INFO

Used in the CMC_DATA_INFO and CMC_RESPONSE_INFO structures.

CMC_TAGGED_OTHER_MSG

Used in the CMC_DATA_INFO and CMC_RESPONSE_INFO structures.

CMC_TAGGED_REQUEST

Used in the CMC_DATA_INFO structures to request a certificate.

CMS_DH_KEY_INFO

Used with the KP_CMS_DH_KEY_INFO parameter in the CryptSetKeyParam function to contain Diffie-Hellman key information.

CMS_KEY_INFO

Not used.

CMSG_CMS_RECIPIENT_INFO

Used with the CryptMsgGetParam function to get information on a key transport, key agreement, or mail list envelope message recipient.

CMSG_CMS_SIGNER_INFO

Contains the content of the defined SignerInfo in signed or signed and enveloped messages.

CMSG_CNG_CONTENT_DECRYPT_INFO

Contains all the relevant information passed between CryptMsgControl and object identifier (OID) installable functions for the import and decryption of a Cryptography API:_Next Generation (CNG) content encryption key (CEK).

CMSG_CONTENT_ENCRYPT_INFO

Contains information shared between the PFN_CMSG_GEN_CONTENT_ENCRYPT_KEY, PFN_CMSG_EXPORT_KEY_TRANS, PFN_CMSG_EXPORT_KEY_AGREE, and PFN_CMSG_EXPORT_MAIL_LIST functions.

CMSG_CTRL_ADD_SIGNER_UNAUTH_ATTR_PARA

Used to add an unauthenticated attribute to a signer of a signed message.

CMSG_CTRL_DECRYPT_PARA

Contains information used to decrypt an enveloped message for a key transport recipient. This structure is passed to CryptMsgControl if the dwCtrlType parameter is CMSG_CTRL_DECRYPT.

CMSG_CTRL_DEL_SIGNER_UNAUTH_ATTR_PARA

Used to delete an unauthenticated attribute of a signer of a signed message.

CMSG_CTRL_KEY_AGREE_DECRYPT_PARA

Contains information about a key agreement recipient.

CMSG_CTRL_KEY_TRANS_DECRYPT_PARA

Contains information about a key transport message recipient.

CMSG_CTRL_MAIL_LIST_DECRYPT_PARA

Contains information on a mail list message recipient.

CMSG_CTRL_VERIFY_SIGNATURE_EX_PARA

Contains information used to verify a message signature. It contains the signer index and signer public key.

CMSG_ENVELOPED_ENCODE_INFO

Contains information needed to encode an enveloped message. It is passed to CryptMsgOpenToEncode if the dwMsgType parameter is CMSG_ENVELOPED.

CMSG_HASHED_ENCODE_INFO

Used with hashed messages. It is passed to the CryptMsgOpenToEncode function if the CryptMsgOpenToEncode function's dwMsgType parameter is CMSG_ENVELOPED.

CMSG_KEY_AGREE_ENCRYPT_INFO

Contains encryption information applicable to all key agreement recipients of an enveloped

message.

CMSG_KEY_AGREE_KEY_ENCRYPT_INFO

Contains the encrypted key for a key agreement recipient of an enveloped message.

CMSG_KEY_AGREE_RECIPIENT_ENCODE_INFO

Contains information about a message recipient that is using key agreement key management.

CMSG_KEY_AGREE_RECIPIENT_INFO

Contains information used for key agreement algorithms.

CMSG_KEY_TRANS_ENCRYPT_INFO

Contains encryption information for a key transport recipient of enveloped data.

CMSG_KEY_TRANS_RECIPIENT_ENCODE_INFO

Contains encoded key transport information for a message recipient.

CMSG_KEY_TRANS_RECIPIENT_INFO

The CMSG_KEY_TRANS_RECIPIENT_INFO structure contains information used in key transport algorithms.

CMSG_MAIL_LIST_ENCRYPT_INFO

Contains encryption information for a mailing list recipient of enveloped data.

CMSG_MAIL_LIST_RECIPIENT_ENCODE_INFO

The CMSG_MAIL_LIST_RECIPIENT_ENCODE_INFO structure is used with previously distributed symmetric keys for decrypting the content key encryption key (KEK).

CMSG_MAIL_LIST_RECIPIENT_INFO

Contains information used for previously distributed symmetric key-encryption keys (KEK).

CMSG_RC2_AUX_INFO

Contains the bit length of the key for RC2 encryption algorithms.

CMSG_RC4_AUX_INFO

The CMSG_RC4_AUX_INFO structure contains the bit length of the key for RC4 encryption algorithms. The pvEncryptionAuxInfo member in CMSG_ENVELOPED_ENCODE_INFO can be set to point to an instance of this structure.

CMSG_RECIPIENT_ENCODE_INFO

Contains information a message recipient's content encryption key management type.

CMSG_RECIPIENT_ENCRYPTED_KEY_ENCODE_INFO

Contains information on a message receiver used to decrypt the session key needed to decrypt the message contents.

CMSG_RECIPIENT_ENCRYPTED_KEY_INFO

The CMSG_RECIPIENT_ENCRYPTED_KEY_INFO structure contains information used for an individual key agreement recipient.

CMSG_SIGNED_ENCODE_INFO

Contains information to be passed to CryptMsgOpenToEncode if dwMsgType is CMSG_SIGNED.

CMSG_SIGNER_ENCODE_INFO

Contains signer information. It is passed to CryptMsgCountersign, CryptMsgCountersignEncoded, and optionally to CryptMsgOpenToEncode as a member of the CMSG_SIGNED_ENCODE_INFO structure, if the dwMsgType parameter is CMSG_SIGNED.

CMSG_SIGNER_INFO

The CMSG_SIGNER_INFO structure contains the content of the PKCS

CMSG_SP3_COMPATIBLE_AUX_INFO

Contains information needed for SP3 compatible encryption.

CMSG_STREAM_INFO

Used to enable stream processing of data rather than single block processing.

CRL_CONTEXT

The CRL_CONTEXT structure contains both the encoded and decoded representations of a certificate revocation list (CRL). CRL contexts returned by any CryptoAPI function must be freed by calling the CertFreeCRLContext function.

CRL_DIST_POINT

Identifies a single certificate revocation list (CRL) distribution point that a certificate user can reference to determine whether certificates have been revoked.

CRL_DIST_POINT_NAME

Identifies a location from which the CRL can be obtained.

CRL_DIST_POINTS_INFO

Contains a list of certificate revocation list (CRL) distribution points a certificate user can reference to determine whether the certificate has been revoked.

CRL_ENTRY

Contains information about a single revoked certificate. It is a member of a CRL_INFO structure.

CRL_FIND_ISSUED_FOR_PARA

Contains the certificate contexts of both a subject and a certificate issuer.

CRL_INFO

Contains the information of a certificate revocation list (CRL).

CRL_ISSUING_DIST_POINT

Contains information about the kinds of certificates listed in a certificate revocation list (CRL).

CROSS_CERT_DIST_POINTS_INFO

Provides information used to update dynamic cross certificates.

CRYPT_AES_128_KEY_STATE

Specifies the 128-bit symmetric key information for an Advanced Encryption Standard (AES) cipher.

CRYPT_AES_256_KEY_STATE

Specifies the 256-bit symmetric key information for an Advanced Encryption Standard (AES) cipher.

CRYPT_ALGORITHM_IDENTIFIER

Specifies an algorithm used to encrypt a private key.

CRYPT_ATTRIBUTE

The CRYPT_ATTRIBUTE structure specifies an attribute that has one or more values.

CRYPT_ATTRIBUTE_TYPE_VALUE

Contains a single attribute value. The Value member's CRYPT_OBJID_BLOB is encoded.

CRYPT_ATTRIBUTES

Contains an array of attributes.

CRYPT_BIT_BLOB

Contains a set of bits represented by an array of bytes.

CRYPT_BLOB_ARRAY

Contains an array of CRYPT_DATA_BLOB structures.

CRYPT_CONTENT_INFO

Contains data encoded in the PKCS

CRYPT_CONTENT_INFO_SEQUENCE_OF_ANY

Contains information representing the Netscape certificate sequence of certificates.

CRYPT_CREDENTIALS

Contains information about credentials that can be passed as optional input to a remote object retrieval function such as CryptRetrieveObjectByUrl or CryptGetTimeValidObject.

CRYPT_DECODE_PARA

Used by the CryptDecodeObjectEx function to provide access to memory allocation and memory freeing callback functions.

CRYPT_DECRYPT_MESSAGE_PARA

The CRYPT_DECRYPT_MESSAGE_PARA structure contains information for decrypting messages.

CRYPT_DEFAULT_CONTEXT_MULTI_OID_PARA

Used with the CryptInstallDefaultContext function to contain an array of object identifier strings.

CRYPT_ECC_CMS_SHARED_INFO

Represents key-encryption key information when using Elliptic Curve Cryptography (ECC) in the Cryptographic Message Syntax (CMS) EnvelopedData content type.

CRYPT_ENCODE_PARA

Used by the CryptEncodeObjectEx function to provide access to memory allocation and memory freeing callback functions.

CRYPT_ENCRYPT_MESSAGE_PARA

Contains information used to encrypt messages.

CRYPT_ENCRYPTED_PRIVATE_KEY_INFO

Contains the information in a PKCS

CRYPT_ENROLLMENT_NAME_VALUE_PAIR

Used to create certificate requests on behalf of a user.

CRYPT_GET_TIME_VALID_OBJECT_EXTRA_INFO

Contains optional extra information that can be passed to the CryptGetTimeValidObject function in the pExtraInfo parameter.

CRYPT_HASH_MESSAGE_PARA

Contains data for hashing messages.

CRYPT_INTEGER_BLOB

The CryptoAPI CRYPT_INTEGER_BLOB structure is used for an arbitrary array of bytes. It is declared in Wincrypt.h and provides flexibility for objects that can contain various data types.

CRYPT_KEY_PROV_INFO

The CRYPT_KEY_PROV_INFO structure contains information about a key container within a cryptographic service provider (CSP).

CRYPT_KEY_PROV_PARAM

Contains information about a key container parameter.

CRYPT_KEY_SIGN_MESSAGE_PARA

Contains information about the cryptographic service provider (CSP) and algorithms used to

sign a message.

CRYPT_KEY_VERIFY_MESSAGE_PARA

Contains information needed to verify signed messages without a certificate for the signer.

CRYPT_MASK_GEN_ALGORITHM

Identifies the algorithm used to generate an RSA PKCS

CRYPT_OBJECT_LOCATOR_PROVIDER_TABLE

Contains pointers to functions implemented by an object location provider.

CRYPT_OID_FUNC_ENTRY

Contains an object identifier (OID) and a pointer to its related function.

CRYPT_OID_INFO

Contains information about an object identifier (OID).

CRYPT_PASSWORD_CREDENTIALSA

Contains the user name and password credentials to be used in the CRYPT_CREDENTIALS structure as optional input to a remote object retrieval function such as CryptRetrieveObjectByUrl or CryptGetTimeValidObject.

CRYPT_PASSWORD_CREDENTIALSW

Contains the user name and password credentials to be used in the CRYPT_CREDENTIALS structure as optional input to a remote object retrieval function such as CryptRetrieveObjectByUrl or CryptGetTimeValidObject.

CRYPT_PKCS12_PBE_PARAMS

Contains parameters used to create an encryption key, initialization vector (IV), or Message Authentication Code (MAC) key for a PKCS

CRYPT_PKCS8_EXPORT_PARAMS

Identifies the private key and a callback function to encrypt the private key. CRYPT_PKCS8_EXPORT_PARAMS is used as a parameter to the CryptExportPKCS8Ex function, which exports a private key in PKCS

CRYPT_PKCS8_IMPORT_PARAMS

Contains a PKCS

CRYPT_PRIVATE_KEY_INFO

Contains a clear-text private key in the PrivateKey field (DER encoded). CRYPT_PRIVATE_KEY_INFO contains the information in a PKCS

CRYPT_PSOURCE_ALGORITHM

Identifies the algorithm and (optionally) the value of the label for an RSAES-OAEP key encryption.

CRYPT_RC2_CBC_PARAMETERS

Contains information used with szOID_RSA_RC2CBC encryption.

CRYPT_RETRIEVE_AUX_INFO

Contains optional information to pass to the CryptRetrieveObjectByUrl function.

CRYPT_RSA_SSA_PSS_PARAMETERS

Contains the parameters for an RSA PKCS

CRYPT_RSAES_OAEP_PARAMETERS

Contains the parameters for an RSAES-OAEP key encryption.

CRYPT_SEQUENCE_OF_ANY

Contains an arbitrary list of encoded BLOBs.

CRYPT_SIGN_MESSAGE_PARA

The CRYPT_SIGN_MESSAGE_PARA structure contains information for signing messages using a specified signing certificate context.

CRYPT_SMART_CARD_ROOT_INFO

Contains the smart card and session IDs associated with a certificate context.

CRYPT_SMIME_CAPABILITIES

Contains a prioritized array of supported capabilities.

CRYPT_SMIME_CAPABILITY

The CRYPT_SMIME_CAPABILITY structure specifies a single capability and its associated parameters. Single capabilities are grouped together into a list of CRYPT_SMIME_CAPABILITIES which can specify a prioritized list of capability preferences.

CRYPT_TIME_STAMP_REQUEST_INFO

Used for time stamping.

CRYPT_TIMESTAMP_ACCURACY

Is used by the CRYPT_TIMESTAMP_INFO structure to represent the accuracy of the time deviation around the UTC time at which the time stamp token was created by the Time Stamp Authority (TSA).

CRYPT_TIMESTAMP_CONTEXT

Contains both the encoded and decoded representations of a time stamp token.

CRYPT_TIMESTAMP_INFO

Contains a signed data content type in Cryptographic Message Syntax (CMS) format.

CRYPT_TIMESTAMP_PARA

Defines additional parameters for the time stamp request.

CRYPT_TIMESTAMP_REQUEST

Defines a time stamp request structure that corresponds to the Abstract Syntax Notation One (ASN.1) definition of a TimeStampReq type.

CRYPT_TIMESTAMP_RESPONSE

Is used internally to encapsulate an Abstract Syntax Notation One (ASN.1) Distinguished Encoding Rules (DER) encoded response.

CRYPT_URL_INFO

Contains information about groupings of URLs.

CRYPT_VERIFY_CERT_SIGN_STRONG_PROPERTIES_INFO

Contains the length, in bits, of the public key and the names of the signing and hashing algorithms used for strong signing.

CRYPT_VERIFY_MESSAGE_PARA

The CRYPT_VERIFY_MESSAGE_PARA structure contains information needed to verify signed messages.

CRYPT_X942_OTHER_INFO

The CRYPT_X942_OTHER_INFO structure contains additional key generation information.

CRYPTNET_URL_CACHE_FLUSH_INFO

Contains expiry information used by the Cryptnet URL Cache (CUC) service to maintain a URL cache entry.

CRYPTNET_URL_CACHE_PRE_FETCH_INFO

Contains update information used by the Cryptnet URL Cache (CUC) service to maintain a URL cache entry.

CRYPTNET_URL_CACHE_RESPONSE_INFO

Contains response information used by the Cryptnet URL Cache (CUC) service to maintain a URL cache entry.

CTL_ANY_SUBJECT_INFO

Contains a SubjectAlgorithm to be matched in the certificate trust list (CTL) and the SubjectIdentifier to be matched in one of the CTL entries in calls to CertFindSubjectInCTL.

CTL_CONTEXT

The CTL_CONTEXT structure contains both the encoded and decoded representations of a CTL.

CTL_ENTRY

An element of a certificate trust list (CTL).

CTL_FIND_SUBJECT_PARA

Contains data used by CertFindCTLInStore with a dwFindType parameter of CTL_FIND_SUBJECT to find a Certificate Trust List (CTL).

CTL_FIND_USAGE_PARA

A member of the CTL_FIND_SUBJECT_PARA structure and it is used by CertFindCTLInStore.

CTL_INFO

Contains the information stored in a Certificate Trust List (CTL).

CTL_USAGE

Contains an array of object identifiers (OIDs) for Certificate Trust List (CTL) extensions.

CTL_USAGE_MATCH

Provides parameters for finding certificate trust lists (CTL) used to build a certificate chain.

CTL_VERIFY_USAGE_PARA

The CTL_VERIFY_USAGE_PARA structure contains parameters used by CertVerifyCTLUsage to establish the validity of a CTL's usage.

CTL_VERIFY_USAGE_STATUS

Contains information about a Certificate Trust List (CTL) returned by CertVerifyCTLUsage.

DHPRIVKEY_VER3

Contains information specific to the particular private key contained in the key BLOB.

DHPUBKEY

Contains information specific to the particular Diffie-Hellman public key contained in the key BLOB.

DHPUBKEY_VER3

Contains information specific to the particular public key contained in the key BLOB.

DSSSEED

Holds the seed and counter values that can be used to verify the primes of the DSS public key.

EV_EXTRA_CERT_CHAIN_POLICY_PARA

Specifies the parameters that are passed in for EV policy validation. Applications use this structure to pass hints to the API that indicate which of the policy qualifier flags of the extended validation certificates are important to the application.

EV_EXTRA_CERT_CHAIN_POLICY_STATUS

Contains policy flags returned from a call to the CertVerifyCertificateChainPolicy function.

HMAC_INFO

The HMAC_INFO structure specifies the hash algorithm and the inner and outer strings that are to be used to calculate the HMAC hash.

HTTPSPolicyCallbackData

Holds policy information used in the verification of Secure Sockets Layer (SSL) client/server certificate chains.

OCSP_BASIC_RESPONSE_ENTRY

Contains the current certificate status for a single certificate.

OCSP_BASIC_RESPONSE_INFO

Contains a basic online certificate status protocol (OCSP) response as specified by RFC 2560.

OCSP_BASIC_REVOKED_INFO

Contains the reason a certificate was revoked.

OCSP_BASIC_SIGNED_RESPONSE_INFO

Contains a basic online certificate status protocol (OCSP) response with a signature.

OCSP_CERT_ID

Contains information to identify a certificate in an online certificate status protocol (OCSP) request or response.

OCSP_REQUEST_ENTRY

Contains information about a single certificate in an online certificate status protocol (OCSP) request.

OCSP_REQUEST_INFO

Contains information for an online certificate status protocol (OCSP) request as specified by RFC 2560.

OCSP_RESPONSE_INFO

Indicates the success or failure of the corresponding online certificate status protocol (OCSP) request. For successful requests, it contains the type and value of response information.

OCSP_SIGNATURE_INFO

Contains a signature for an online certificate status protocol (OCSP) request or response.

OCSP_SIGNED_REQUEST_INFO

Contains information for an online certificate status protocol (OCSP) request with optional signature information.

PROV_ENUMALGS

Used with the CryptGetProvParam function when the PP_ENUMALGS parameter is retrieved to contain information about an algorithm supported by a cryptographic service provider (CSP).

PROV_ENUMALGS_EX

Used with the CryptGetProvParam function when the PP_ENUMALGS_EX parameter is retrieved to contain information about an algorithm supported by a cryptographic service provider (CSP).

ROOT_INFO_LUID

Contains a locally unique identifier (LUID) for Cryptographic Smart Card Root Information.

RSAPUBKEY

The RSAPUBKEY structure contains information specific to the particular public key contained in the key BLOB.

SCHANNEL_ALG

The SCHANNEL_ALG structure contains algorithm and key size information. It is used as the structure passed as pbData in CryptSetKeyParam when dwParam is set to KP_SCHANNEL_ALG.

SSL_F12_EXTRA_CERT_CHAIN_POLICY_STATUS

The SSL_F12_EXTRA_CERT_CHAIN_POLICY_STATUS structure checks if any certificates in the chain have weak cryptography and checks if a third party root certificate is compliant with the Microsoft Root Program requirements.

Is this page helpful?

Yes No

EXHIBIT H

Server Core Functions by DLL

10/03/2017 • 60 minutes to read

This list contains, by the associated DLL, the APIs supported in full by Server Core. Other APIs may be provided via the DLLs included with Server Core, and depending on how they are called, may operate without error. However, when APIs not listed in this documentation are utilized and the code does not have any dependencies outside of the DLLs included in Server Core, it is highly recommended that the code is tested to ensure it works properly.

DLL Name	Function Name
activeds.dll	ADsBuildEnumerator
activeds.dll	ADsBuildVarArrayInt
activeds.dll	ADsBuildVarArrayStr
activeds.dll	ADsEncodeBinaryData
activeds.dll	ADsEnumerateNext
activeds.dll	ADsFreeEnumerator
activeds.dll	ADsGetLastError
activeds.dll	ADsGetObject
activeds.dll	ADsSetLastError
activeds.dll	AllocADsMem
activeds.dll	AllocADsStr

activeds.dll	FreeADsMem
activeds.dll	FreeADsStr
activeds.dll	ReallocADsMem
activeds.dll	ReallocADsStr
advapi32.dll	AbortSystemShutdownA
advapi32.dll	AbortSystemShutdownW
advapi32.dll	AccessCheck
advapi32.dll	AccessCheckAndAuditAlarmA
advapi32.dll	AccessCheckAndAuditAlarmW
advapi32.dll	AccessCheckByType
advapi32.dll	AccessCheckByTypeAndAuditAlarmA
advapi32.dll	AccessCheckByTypeAndAuditAlarmW
advapi32.dll	AccessCheckByTypeResultList
advapi32.dll	AccessCheckByTypeResultListAndAuditAlarmA
advapi32.dll	AccessCheckByTypeResultListAndAuditAlarmByHandleA
advapi32.dll	AccessCheckByTypeResultListAndAuditAlarmByHandleW
advapi32.dll	AccessCheckByTypeResultListAndAuditAlarmW
advapi32.dll	AddAccessAllowedAce

advapi32.dll	AddAccessAllowedAceEx
advapi32.dll	AddAccessAllowedObjectAce
advapi32.dll	AddAccessDeniedAce
advapi32.dll	AddAccessDeniedAceEx
advapi32.dll	AddAccessDeniedObjectAce
advapi32.dll	AddAce
advapi32.dll	AddAuditAccessAce
advapi32.dll	AddAuditAccessAceEx
advapi32.dll	AddAuditAccessObjectAce
advapi32.dll	AddMandatoryAce
advapi32.dll	AdjustTokenGroups
advapi32.dll	AdjustTokenPrivileges
advapi32.dll	AllocateAndInitializeSid
advapi32.dll	AllocateLocallyUniqueId
advapi32.dll	AreAllAccessesGranted
advapi32.dll	AreAnyAccessesGranted
advapi32.dll	AuditComputeEffectivePolicyBySid
advapi32.dll	AuditComputeEffectivePolicyByToken
advapi32.dll	AuditEnumerateCategories

advapi32.dll	AuditEnumeratePerUserPolicy
advapi32.dll	AuditEnumerateSubCategories
advapi32.dll	AuditFree
advapi32.dll	AuditLookupCategoryGuidFromCategoryId
advapi32.dll	AuditLookupCategoryIdFromCategoryGuid
advapi32.dll	AuditLookupCategoryNameA
advapi32.dll	AuditLookupCategoryNameW
advapi32.dll	AuditLookupSubCategoryNameA
advapi32.dll	AuditLookupSubCategoryNameW
advapi32.dll	AuditQueryPerUserPolicy
advapi32.dll	AuditQuerySecurity
advapi32.dll	AuditQuerySystemPolicy
advapi32.dll	AuditSetPerUserPolicy
advapi32.dll	AuditSetSecurity
advapi32.dll	AuditSetSystemPolicy
advapi32.dll	BackupEventLogA
advapi32.dll	BackupEventLogW
advapi32.dll	BuildExplicitAccessWithNameA

advapi32.dll	BuildTrusteeWithNameA
advapi32.dll	BuildTrusteeWithObjectsAndNameA
advapi32.dll	BuildTrusteeWithObjectsAndSidA
advapi32.dll	BuildTrusteeWithSidA
advapi32.dll	ChangeServiceConfig2A
advapi32.dll	ChangeServiceConfig2W
advapi32.dll	ChangeServiceConfigA
advapi32.dll	ChangeServiceConfigW
advapi32.dll	CheckTokenMembership
advapi32.dll	ClearEventLogA
advapi32.dll	ClearEventLogW
advapi32.dll	CloseEncryptedFileRaw
advapi32.dll	CloseEventLog
advapi32.dll	CloseServiceHandle
advapi32.dll	CloseThreadWaitChainSession
advapi32.dll	CloseTrace
advapi32.dll	ControlService
advapi32.dll	ControlServiceExA
advapi32.dll	ControlServiceExW

advapi32.dll	ConvertSidToStringSidA
advapi32.dll	ConvertSidToStringSidW
advapi32.dll	ConvertToAutoInheritPrivateObjectSecurity
advapi32.dll	CopySid
advapi32.dll	CreatePrivateObjectSecurity
advapi32.dll	CreatePrivateObjectSecurityEx
advapi32.dll	CreatePrivateObjectSecurityWithMultipleInheritance
advapi32.dll	CreateProcessAsUserA
advapi32.dll	CreateProcessAsUserW
advapi32.dll	CreateProcessWithLogonW
advapi32.dll	CreateProcessWithTokenW
advapi32.dll	CreateRestrictedToken
advapi32.dll	CreateServiceA
advapi32.dll	CreateServiceW
advapi32.dll	CreateTraceInstanceId
advapi32.dll	CreateWellKnownSid
advapi32.dll	CredDeleteA
advapi32.dll	CredDeleteW

advapi32.dll	CredEnumerateA
advapi32.dll	CredEnumerateW
advapi32.dll	CredFindBestCredentialA
advapi32.dll	CredFindBestCredentialW
advapi32.dll	CredGetSessionTypes
advapi32.dll	CredGetTargetInfoA
advapi32.dll	CredGetTargetInfoW
advapi32.dll	CredIsMarshaledCredentialA
advapi32.dll	CredIsMarshaledCredentialW
advapi32.dll	CredIsProtectedA
advapi32.dll	CredIsProtectedW
advapi32.dll	CredMarshalCredentialA
advapi32.dll	CredMarshalCredentialW
advapi32.dll	CredProtectA
advapi32.dll	CredProtectW
advapi32.dll	CredReadA
advapi32.dll	CredReadDomainCredentialsA
advapi32.dll	CredReadDomainCredentialsW
advapi32.dll	CredReadW

advapi32.dll	CredRenameA
advapi32.dll	CredUnmarshalCredentialA
advapi32.dll	CredUnmarshalCredentialW
advapi32.dll	CredUnprotectA
advapi32.dll	CredUnprotectW
advapi32.dll	CredWriteA
advapi32.dll	CredWriteDomainCredentialsA
advapi32.dll	CredWriteDomainCredentialsW
advapi32.dll	CredWriteW
advapi32.dll	CryptAcquireContextA
advapi32.dll	CryptAcquireContextW
advapi32.dll	CryptContextAddRef
advapi32.dll	CryptCreateHash
advapi32.dll	CryptDecrypt
advapi32.dll	CryptDeriveKey
advapi32.dll	CryptDestroyHash
advapi32.dll	CryptDestroyKey
advapi32.dll	CryptDuplicateHash

advapi32.dll	CryptDuplicateKey
advapi32.dll	CryptEncrypt
advapi32.dll	CryptEnumProvidersA
advapi32.dll	CryptEnumProvidersW
advapi32.dll	CryptEnumProviderTypesA
advapi32.dll	CryptEnumProviderTypesW
advapi32.dll	CryptExportKey
advapi32.dll	CryptGenKey
advapi32.dll	CryptGenRandom
advapi32.dll	CryptGetDefaultProviderA
advapi32.dll	CryptGetDefaultProviderW
advapi32.dll	CryptGetHashParam
advapi32.dll	CryptGetKeyParam
advapi32.dll	CryptGetProvParam
advapi32.dll	CryptGetUserKey
advapi32.dll	CryptHashData
advapi32.dll	CryptHashSessionKey
advapi32.dll	CryptImportKey
advapi32.dll	CryptReleaseContext

advapi32.dll	CryptSetHashParam
advapi32.dll	CryptSetKeyParam
advapi32.dll	CryptSetProviderA
advapi32.dll	CryptSetProviderExA
advapi32.dll	CryptSetProviderExW
advapi32.dll	CryptSetProviderW
advapi32.dll	CryptSetProvParam
advapi32.dll	CryptSignHashA
advapi32.dll	CryptSignHashW
advapi32.dll	CryptVerifySignatureA
advapi32.dll	CryptVerifySignatureW
advapi32.dll	DeleteAce
advapi32.dll	DeleteService
advapi32.dll	DeregisterEventSource
advapi32.dll	DestroyPrivateObjectSecurity
advapi32.dll	DuplicateToken
advapi32.dll	DuplicateTokenEx
advapi32.dll	EnumDependentServicesA

advapi32.dll	EnumDependentServicesW
advapi32.dll	EnumerateTraceGuids
advapi32.dll	EnumerateTraceGuidsEx
advapi32.dll	EnumServicesStatusA
advapi32.dll	EnumServicesStatusExA
advapi32.dll	EnumServicesStatusExW
advapi32.dll	EnumServicesStatusW
advapi32.dll	EqualDomainSid
advapi32.dll	EqualPrefixSid
advapi32.dll	EqualSid
advapi32.dll	EventAccessQuery
advapi32.dll	EventAccessRemove
advapi32.dll	EventActivityIdControl
advapi32.dll	EventEnabled
advapi32.dll	EventProviderEnabled
advapi32.dll	EventRegister
advapi32.dll	EventUnregister
advapi32.dll	EventWrite
advapi32.dll	EventWriteEx

advapi32.dll	EventWriteString
advapi32.dll	EventWriteTransfer
advapi32.dll	FindFirstFreeAce
advapi32.dll	FreeEncryptionCertificateHashList
advapi32.dll	FreeSid
advapi32.dll	GetAce
advapi32.dll	GetAclInformation
advapi32.dll	GetCurrentHwProfileA
advapi32.dll	GetCurrentHwProfileW
advapi32.dll	GetEventLogInformation
advapi32.dll	GetFileSecurityA
advapi32.dll	GetFileSecurityW
advapi32.dll	GetInheritanceSourceA
advapi32.dll	GetKernelObjectSecurity
advapi32.dll	GetLengthSid
advapi32.dll	GetLocalManagedApplications
advapi32.dll	GetNumberOfEventLogRecords
advapi32.dll	GetOldestEventLogRecord

advapi32.dll	GetPrivateObjectSecurity
advapi32.dll	GetSecurityDescriptorControl
advapi32.dll	GetSecurityDescriptorDacl
advapi32.dll	GetSecurityDescriptorGroup
advapi32.dll	GetSecurityDescriptorLength
advapi32.dll	GetSecurityDescriptorOwner
advapi32.dll	GetSecurityDescriptorRMControl
advapi32.dll	GetSecurityDescriptorSacl
advapi32.dll	GetServiceDisplayNameA
advapi32.dll	GetServiceKeyNameA
advapi32.dll	GetSidIdentifierAuthority
advapi32.dll	GetSidLengthRequired
advapi32.dll	GetSidSubAuthority
advapi32.dll	GetSidSubAuthorityCount
advapi32.dll	GetTokenInformation
advapi32.dll	GetTraceEnableFlags
advapi32.dll	GetTraceEnableLevel
advapi32.dll	GetTraceLoggerHandle
advapi32.dll	GetTrusteeFormA

advapi32.dll	GetTrusteeNameA
advapi32.dll	GetTrusteeTypeA
advapi32.dll	GetUserNameA
advapi32.dll	GetUserNameW
advapi32.dll	GetWindowsAccountDomainSid
advapi32.dll	ImpersonateAnonymousToken
advapi32.dll	ImpersonateLoggedOnUser
advapi32.dll	ImpersonateNamedPipeClient
advapi32.dll	ImpersonateSelf
advapi32.dll	InitializeAcl
advapi32.dll	InitializeSecurityDescriptor
advapi32.dll	InitializeSid
advapi32.dll	InitiateShutdownA
advapi32.dll	InitiateShutdownW
advapi32.dll	InitiateSystemShutdownA
advapi32.dll	InitiateSystemShutdownExA
advapi32.dll	InitiateSystemShutdownExW
advapi32.dll	InitiateSystemShutdownW

advapi32.dll	IsTextUnicode
advapi32.dll	IsTokenRestricted
advapi32.dll	IsValidAcl
advapi32.dll	IsValidSecurityDescriptor
advapi32.dll	IsValidSid
advapi32.dll	IsWellKnownSid
advapi32.dll	LockServiceDatabase
advapi32.dll	LogonUserExExW
advapi32.dll	LookupAccountNameA
advapi32.dll	LookupAccountNameW
advapi32.dll	LookupAccountSidA
advapi32.dll	LookupAccountSidW
advapi32.dll	LookupPrivilegeDisplayNameA
advapi32.dll	LookupPrivilegeDisplayNameW
advapi32.dll	LookupPrivilegeNameA
advapi32.dll	LookupPrivilegeNameW
advapi32.dll	LookupPrivilegeValueA
advapi32.dll	LookupPrivilegeValueW
advapi32.dll	LsaAddAccountRights

advapi32.dll	LsaClose
advapi32.dll	LsaCreateTrustedDomainEx
advapi32.dll	LsaDeleteTrustedDomain
advapi32.dll	LsaEnumerateAccountRights
advapi32.dll	LsaEnumerateAccountsWithUserRight
advapi32.dll	LsaEnumerateTrustedDomains
advapi32.dll	LsaEnumerateTrustedDomainsEx
advapi32.dll	LsaFreeMemory
advapi32.dll	LsaLookupNames
advapi32.dll	LsaLookupNames2
advapi32.dll	LsaLookupSids
advapi32.dll	LsaNtStatusToWinError
advapi32.dll	LsaOpenPolicy
advapi32.dll	LsaOpenTrustedDomainByName
advapi32.dll	LsaQueryDomainInformationPolicy
advapi32.dll	LsaQueryForestTrustInformation
advapi32.dll	LsaQueryInformationPolicy
advapi32.dll	LsaQueryTrustedDomainInfo

advapi32.dll	LsaQueryTrustedDomainInfoByName
advapi32.dll	LsaRemoveAccountRights
advapi32.dll	LsaRetrievePrivateData
advapi32.dll	LsaSetDomainInformationPolicy
advapi32.dll	LsaSetForestTrustInformation
advapi32.dll	LsaSetInformationPolicy
advapi32.dll	LsaSetTrustedDomainInfoByName
advapi32.dll	LsaSetTrustedDomainInformation
advapi32.dll	LsaStorePrivateData
advapi32.dll	MakeAbsoluteSD
advapi32.dll	MakeSelfRelativeSD
advapi32.dll	MapGenericMask
advapi32.dll	MSChapSrvChangePassword
advapi32.dll	MSChapSrvChangePassword2
advapi32.dll	NotifyBootConfigStatus
advapi32.dll	NotifyChangeEventLog
advapi32.dll	NotifyServiceStatusChange
advapi32.dll	NotifyServiceStatusChangeA
advapi32.dll	NotifyServiceStatusChangeW

advapi32.dll	ObjectCloseAuditAlarmA
advapi32.dll	ObjectCloseAuditAlarmW
advapi32.dll	ObjectDeleteAuditAlarmA
advapi32.dll	ObjectDeleteAuditAlarmW
advapi32.dll	ObjectOpenAuditAlarmA
advapi32.dll	ObjectOpenAuditAlarmW
advapi32.dll	ObjectPrivilegeAuditAlarmA
advapi32.dll	ObjectPrivilegeAuditAlarmW
advapi32.dll	OpenBackupEventLogA
advapi32.dll	OpenBackupEventLogW
advapi32.dll	OpenEventLogA
advapi32.dll	OpenEventLogW
advapi32.dll	OpenProcessToken
advapi32.dll	OpenSCManagerA
advapi32.dll	OpenSCManagerW
advapi32.dll	OpenServiceA
advapi32.dll	OpenServiceW
advapi32.dll	OpenThreadToken

advapi32.dll	OpenThreadWaitChainSession
advapi32.dll	PerfCreateInstance
advapi32.dll	PerfDecrementULongCounterValue
advapi32.dll	PerfDecrementULongLongCounterValue
advapi32.dll	PerfDeleteInstance
advapi32.dll	PerfIncrementULongCounterValue
advapi32.dll	PerfIncrementULongLongCounterValue
advapi32.dll	PerfQueryInstance
advapi32.dll	PerfSetCounterRefValue
advapi32.dll	PerfSetCounterSetInfo
advapi32.dll	PerfSetULongCounterValue
advapi32.dll	PerfSetULongLongCounterValue
advapi32.dll	PerfStartProvider
advapi32.dll	PerfStartProviderEx
advapi32.dll	PerfStopProvider
advapi32.dll	PrivilegeCheck
advapi32.dll	PrivilegedServiceAuditAlarmA
advapi32.dll	PrivilegedServiceAuditAlarmW
advapi32.dll	QuerySecurityAccessMask

advapi32.dll	QueryServiceConfig2A
advapi32.dll	QueryServiceConfig2W
advapi32.dll	QueryServiceConfigA
advapi32.dll	QueryServiceConfigW
advapi32.dll	QueryServiceLockStatusA
advapi32.dll	QueryServiceLockStatusW
advapi32.dll	QueryServiceObjectSecurity
advapi32.dll	QueryServiceStatus
advapi32.dll	QueryServiceStatusEx
advapi32.dll	ReadEncryptedFileRaw
advapi32.dll	ReadEventLogA
advapi32.dll	ReadEventLogW
advapi32.dll	RegCloseKey
advapi32.dll	RegConnectRegistryA
advapi32.dll	RegConnectRegistryW
advapi32.dll	RegCopyTreeA
advapi32.dll	RegCopyTreeW
advapi32.dll	RegCreateKeyA

advapi32.dll	RegCreateKeyExA
advapi32.dll	RegCreateKeyExW
advapi32.dll	RegCreateKeyTransactedA
advapi32.dll	RegCreateKeyTransactedW
advapi32.dll	RegCreateKeyW
advapi32.dll	RegDeleteKeyA
advapi32.dll	RegDeleteKeyExA
advapi32.dll	RegDeleteKeyExW
advapi32.dll	RegDeleteKeyTransactedA
advapi32.dll	RegDeleteKeyTransactedW
advapi32.dll	RegDeleteKeyValueA
advapi32.dll	RegDeleteKeyValueW
advapi32.dll	RegDeleteKeyW
advapi32.dll	RegDeleteTreeA
advapi32.dll	RegDeleteTreeW
advapi32.dll	RegDeleteValueA
advapi32.dll	RegDeleteValueW
advapi32.dll	RegDisablePredefinedCache
advapi32.dll	RegDisablePredefinedCacheEx

advapi32.dll	RegDisableReflectionKey
advapi32.dll	RegEnumKeyA
advapi32.dll	RegEnumKeyExA
advapi32.dll	RegEnumKeyExW
advapi32.dll	RegEnumKeyW
advapi32.dll	RegEnumValueA
advapi32.dll	RegEnumValueW
advapi32.dll	RegFlushKey
advapi32.dll	RegGetKeySecurity
advapi32.dll	RegGetValueA
advapi32.dll	RegGetValueW
advapi32.dll	RegisterEventSourceA
advapi32.dll	RegisterEventSourceW
advapi32.dll	RegisterServiceCtrlHandlerA
advapi32.dll	RegisterServiceCtrlHandlerExA
advapi32.dll	RegisterServiceCtrlHandlerExW
advapi32.dll	RegisterServiceCtrlHandlerW
advapi32.dll	RegisterTraceGuidsA

advapi32.dll	RegisterTraceGuidsW
advapi32.dll	RegisterWaitChainCOMCallback
advapi32.dll	RegLoadAppKeyA
advapi32.dll	RegLoadAppKeyW
advapi32.dll	RegLoadKeyA
advapi32.dll	RegLoadKeyW
advapi32.dll	RegLoadMUIStringA
advapi32.dll	RegLoadMUIStringW
advapi32.dll	RegNotifyChangeKeyValue
advapi32.dll	RegOpenCurrentUser
advapi32.dll	RegOpenKeyA
advapi32.dll	RegOpenKeyExA
advapi32.dll	RegOpenKeyExW
advapi32.dll	RegOpenKeyTransactedA
advapi32.dll	RegOpenKeyTransactedW
advapi32.dll	RegOpenKeyW
advapi32.dll	RegOpenUserClassesRoot
advapi32.dll	RegOverridePredefKey
advapi32.dll	RegQueryInfoKeyA

advapi32.dll	RegQueryInfoKeyW
advapi32.dll	RegQueryMultipleValuesA
advapi32.dll	RegQueryMultipleValuesW
advapi32.dll	RegQueryReflectionKey
advapi32.dll	RegQueryValueA
advapi32.dll	RegQueryValueExA
advapi32.dll	RegQueryValueExW
advapi32.dll	RegQueryValueW
advapi32.dll	RegReplaceKeyA
advapi32.dll	RegReplaceKeyW
advapi32.dll	RegRestoreKeyA
advapi32.dll	RegRestoreKeyW
advapi32.dll	RegSaveKeyA
advapi32.dll	RegSaveKeyExA
advapi32.dll	RegSaveKeyExW
advapi32.dll	RegSaveKeyW
advapi32.dll	RegSetKeySecurity
advapi32.dll	RegSetKeyValueA

advapi32.dll	RegSetKeyValueW
advapi32.dll	RegSetValueA
advapi32.dll	RegSetValueExA
advapi32.dll	RegSetValueExW
advapi32.dll	RegSetValueW
advapi32.dll	RegUnLoadKeyA
advapi32.dll	RegUnLoadKeyW
advapi32.dll	RemoveTraceCallback
advapi32.dll	ReportEventA
advapi32.dll	ReportEventW
advapi32.dll	RevertToSelf
advapi32.dll	SaferCloseLevel
advapi32.dll	SaferComputeTokenFromLevel
advapi32.dll	SaferCreateLevel
advapi32.dll	SaferGetPolicyInformation
advapi32.dll	SaferiIsExecutableFileType
advapi32.dll	SaferiSearchMatchingHashRules
advapi32.dll	SetAclInformation

advapi32.dll	SetFileSecurityA
advapi32.dll	SetFileSecurityW
advapi32.dll	SetKernelObjectSecurity
advapi32.dll	SetPrivateObjectSecurity
advapi32.dll	SetPrivateObjectSecurityEx
advapi32.dll	SetSecurityAccessMask
advapi32.dll	SetSecurityDescriptorControl
advapi32.dll	SetSecurityDescriptorDacl
advapi32.dll	SetSecurityDescriptorGroup
advapi32.dll	SetSecurityDescriptorOwner
advapi32.dll	SetSecurityDescriptorRMControl
advapi32.dll	SetSecurityDescriptorSacl
advapi32.dll	SetServiceBits
advapi32.dll	SetServiceObjectSecurity
advapi32.dll	SetServiceStatus
advapi32.dll	SetThreadToken
advapi32.dll	SetTokenInformation
advapi32.dll	SetTraceCallback
advapi32.dll	StartServiceA

advapi32.dll	StartServiceCtrlDispatcherA
advapi32.dll	StartServiceCtrlDispatcherW
advapi32.dll	StartServiceW
advapi32.dll	TraceEvent
advapi32.dll	TraceEventInstance
advapi32.dll	TraceMessage
advapi32.dll	TraceMessageVa
advapi32.dll	TraceSetInformation
advapi32.dll	TreeResetNamedSecurityInfoA
advapi32.dll	UnlockServiceDatabase
advapi32.dll	UnregisterTraceGuids
advapi32.dll	WriteEncryptedFileRaw
authz.dll	AuthzAddSidsToContext
authz.dll	AuthzEnumerateSecurityEventSources
authz.dll	AuthzFreeAuditEvent
authz.dll	AuthzFreeContext
authz.dll	AuthzFreeHandle
authz.dll	AuthzFreeResourceManager

authz.dll	AuthzGetInformationFromContext
authz.dll	AuthzInitializeContextFromAuthzContext
authz.dll	AuthzInitializeContextFromSid
authz.dll	AuthzInitializeContextFromToken
authz.dll	AuthzInitializeObjectAccessAuditEvent
authz.dll	AuthzInitializeObjectAccessAuditEvent2
authz.dll	AuthzInitializeResourceManager
authz.dll	AuthzInstallSecurityEventSource
authz.dll	AuthzModifySecurityAttributes
authz.dll	AuthzRegisterSecurityEventSource
authz.dll	AuthzReportSecurityEvent
authz.dll	AuthzReportSecurityEventFromParams
authz.dll	AuthzUninstallSecurityEventSource
authz.dll	AuthzUnregisterSecurityEventSource
bcrypt.dll	BCryptAddContextFunction
bcrypt.dll	BCryptAddContextFunctionProvider
bcrypt.dll	BCryptCloseAlgorithmProvider
bcrypt.dll	BCryptConfigureContext
bcrypt.dll	BCryptConfigureContextFunction

bcrypt.dll	BCryptCreateContext
bcrypt.dll	BCryptCreateHash
bcrypt.dll	BCryptDecrypt
bcrypt.dll	BCryptDeleteContext
bcrypt.dll	BCryptDeriveKey
bcrypt.dll	BCryptDeriveKeyCapi
bcrypt.dll	BCryptDeriveKeyPBKDF2
bcrypt.dll	BCryptDestroyHash
bcrypt.dll	BCryptDestroyKey
bcrypt.dll	BCryptDestroySecret
bcrypt.dll	BCryptDuplicateHash
bcrypt.dll	BCryptDuplicateKey
bcrypt.dll	BCryptEncrypt
bcrypt.dll	BCryptEnumAlgorithms
bcrypt.dll	BCryptEnumContextFunctionProviders
bcrypt.dll	BCryptEnumContextFunctions
bcrypt.dll	BCryptEnumContexts
bcrypt.dll	BCryptEnumProviders

bcrypt.dll	BCryptEnumRegisteredProviders
bcrypt.dll	BCryptExportKey
bcrypt.dll	BCryptFinalizeKeyPair
bcrypt.dll	BCryptFinishHash
bcrypt.dll	BCryptFreeBuffer
bcrypt.dll	BCryptGenerateKeyPair
bcrypt.dll	BCryptGenerateSymmetricKey
bcrypt.dll	BCryptGenRandom
bcrypt.dll	BCryptGetFipsAlgorithmMode
bcrypt.dll	BCryptGetProperty
bcrypt.dll	BCryptHashData
bcrypt.dll	BCryptImportKey
bcrypt.dll	BCryptImportKeyPair
bcrypt.dll	BCryptOpenAlgorithmProvider
bcrypt.dll	BCryptQueryContextConfiguration
bcrypt.dll	BCryptQueryContextFunctionConfiguration
bcrypt.dll	BCryptQueryContextFunctionProperty
bcrypt.dll	BCryptQueryProviderRegistration

bcrypt.dll	BCryptRegisterConfigChangeNotify
bcrypt.dll	BCryptRemoveContextFunction
bcrypt.dll	BCryptRemoveContextFunctionProvider
bcrypt.dll	BCryptResolveProviders
bcrypt.dll	BCryptSecretAgreement
bcrypt.dll	BCryptSetContextFunctionProperty
bcrypt.dll	BCryptSetProperty
bcrypt.dll	BCryptSignHash
bcrypt.dll	BCryptUnregisterConfigChangeNotify
bcrypt.dll	BCryptVerifySignature
cabinet.dll	DeleteExtractedFiles
cabinet.dll	DllGetVersion
cabinet.dll	Extract
cabinet.dll	FCIAddFile
cabinet.dll	FCICreate
cabinet.dll	FCIDestroy
cabinet.dll	FCIFlushCabinet
cabinet.dll	FCIFlushFolder
cabinet.dll	FDICopy

crypt32.dll	CertFindExtension
crypt32.dll	CertFindRDNAttr
crypt32.dll	CertFindSubjectInSortedCTL
crypt32.dll	CertFreeCertificateChainList
crypt32.dll	CertGetServerOcspResponseContext
crypt32.dll	CertRDNValueToStrA
crypt32.dll	CertRDNValueToStrW
crypt32.dll	CertSerializeCRLStoreElement
crypt32.dll	CertSetStoreProperty
crypt32.dll	CertVerifyCRLRevocation
crypt32.dll	CertVerifyCRLTimeValidity
crypt32.dll	CertVerifyTimeValidity
crypt32.dll	CertVerifyValidityNesting
crypt32.dll	CryptBinaryToStringA
crypt32.dll	CryptBinaryToStringW
crypt32.dll	CryptEnumOIDFunction
crypt32.dll	CryptFindLocalizedName
crypt32.dll	CryptGetDefaultOIDDllList
crypt32.dll	CryptGetOIDFunctionValue

crypt32.dll	CryptHashCertificate
crypt32.dll	CryptInitOIDFunctionSet
crypt32.dll	CryptInstallDefaultContext
crypt32.dll	CryptMemAlloc
crypt32.dll	CryptMemRealloc
crypt32.dll	CryptMsgClose
crypt32.dll	CryptMsgDuplicate
crypt32.dll	CryptMsgOpenToDecode
crypt32.dll	CryptProtectMemory
crypt32.dll	CryptRegisterOIDFunction
crypt32.dll	CryptRegisterOIDInfo
crypt32.dll	CryptSetOIDFunctionValue
crypt32.dll	CryptSIPAddProvider
crypt32.dll	CryptSIPRemoveProvider
crypt32.dll	CryptSIPRetrieveSubjectGuidForCatalogFile
crypt32.dll	CryptStringToBinaryA
crypt32.dll	CryptStringToBinaryW
crypt32.dll	CryptUninstallDefaultContext

crypt32.dll	CryptUnprotectMemory
crypt32.dll	CryptUnregisterOIDFunction
crypt32.dll	CryptUnregisterOIDInfo
crypt32.dll	CryptUpdateProtectedState
crypt32.dll	PFXIsPFXBlob
cryptdll.dll	MD5Final
cryptdll.dll	MD5Init
cryptdll.dll	MD5Update
cryptnet.dll	CryptGetObjectUrl
cryptnet.dll	CryptGetTimeValidObject
cryptxml.dll	CryptXmlClose
cryptxml.dll	CryptXmlGetTransforms
cscapi.dll	OfflineFilesEnable
cscapi.dll	OfflineFilesQueryStatus
dbghelp.dll	EnumDirTree
dbghelp.dll	EnumDirTreeW
dbghelp.dll	EnumerateLoadedModules
dbghelp.dll	EnumerateLoadedModules64

mswsock.dll	TransmitFile
ncrypt.dll	NCryptCreatePersistedKey
ncrypt.dll	NCryptDecrypt
ncrypt.dll	NCryptDeleteKey
ncrypt.dll	NCryptDeriveKey
ncrypt.dll	NCryptEncrypt
ncrypt.dll	NCryptEnumAlgorithms
ncrypt.dll	NCryptEnumKeys
ncrypt.dll	NCryptEnumStorageProviders
ncrypt.dll	NCryptExportKey
ncrypt.dll	NCryptFinalizeKey
ncrypt.dll	NCryptFreeBuffer
ncrypt.dll	NCryptFreeObject
ncrypt.dll	NCryptGetProperty
ncrypt.dll	NCryptImportKey
ncrypt.dll	NCryptIsAlgSupported
ncrypt.dll	NCryptIsKeyHandle
ncrypt.dll	NCryptNotifyChangeKey

ncrypt.dll	NCryptOpenKey
ncrypt.dll	NCryptOpenStorageProvider
ncrypt.dll	NCryptSecretAgreement
ncrypt.dll	NCryptSetProperty
ncrypt.dll	NCryptSignHash
ncrypt.dll	NCryptTranslateHandle
ncrypt.dll	NCryptVerifySignature
ndfapi.dll	NdfCloseIncident
ndfapi.dll	NdfExecuteDiagnosis
ndis.sys	NdisAdjustBufferLength
ndis.sys	NdisAdvanceNetBufferDataStart
ndis.sys	NdisAdvanceNetBufferListDataStart
ndis.sys	NdisAllocateBufferPool
ndis.sys	NdisAllocateSpinLock
ndis.sys	NdisBufferLength
ndis.sys	NdisCancelSendPackets
ndis.sys	NdisClGetProtocolVcContextFromTapiCallId
ndis.sys	NdisClModifyCallQoS
ndis.sys	NdisCmDispatchCallConnected

EXHIBIT I

Microsoft Windows

FIPS 140 Validation

Microsoft Windows 10 (Creators Update)

Microsoft Windows 10 Mobile (Creators Update)

Non-Proprietary

Security Policy Document

Version Number	1.01
Updated On	March 15, 2018

Version History

Version	Date	Summary of changes
1.0	October 3, 2017	Draft sent to NIST CMVP
1.01	February 12, 2018	Updates for CMVP

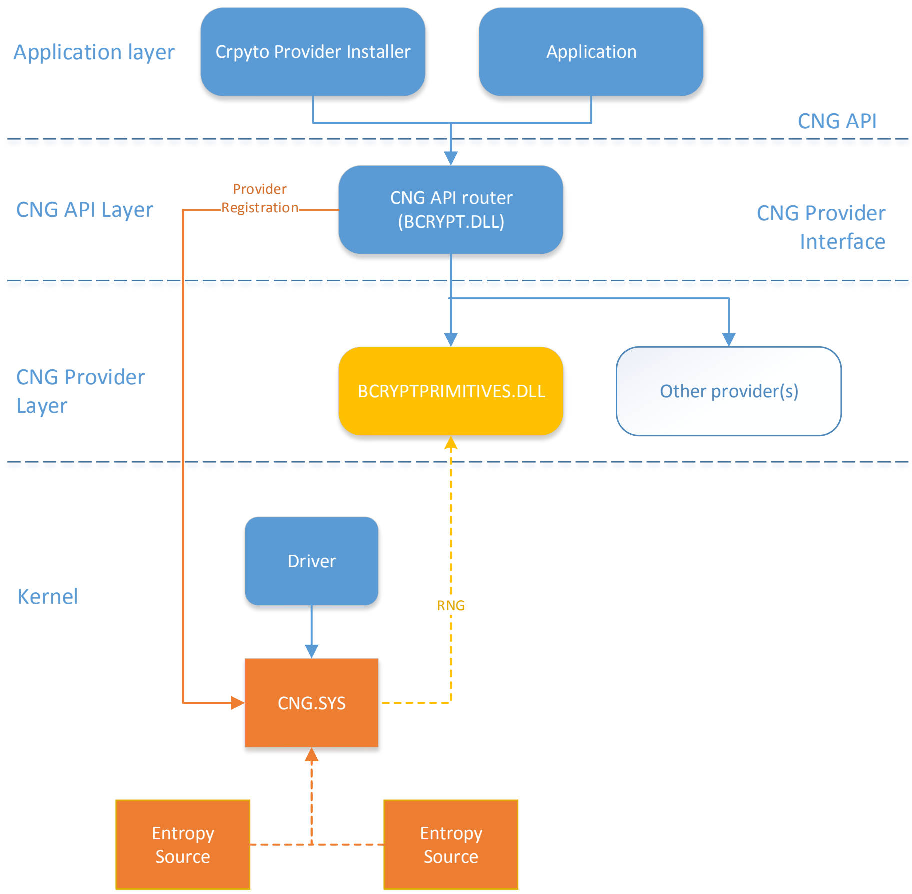
TABLE OF CONTENTS

1 Introduction

Microsoft Kernel Mode Cryptographic Primitives Library is a kernel-mode cryptographic module that provides cryptographic services through the Microsoft CNG (Cryptography, Next Generation) API to Windows 10 kernel components.

Kernel Mode Cryptographic Primitives Library also provides cryptographic provider registration and configuration services to both user and kernel mode components. See Non-Security Relevant Configuration Interfaces for more information.

The relationship between Kernel Mode Cryptographic Primitives Library and other components is shown in the following diagram:



1.1 List of Cryptographic Module Binary Executables

The Kernel Mode Cryptographic Primitives Library consists of the following binaries:

- CNG.SYS

The Windows build covered by this validation is:

Version 10.0.15063

1.2 Validated Platforms

The Windows editions covered by this validation are:

- Windows 10 Home Edition (32-bit version)
- Windows 10 Pro Edition (64-bit version)
- Windows 10 Enterprise Edition (64-bit version)
- Windows 10 Education Edition (64-bit version)
- Windows 10 S Edition (64-bit version)
- Windows 10 Mobile
- Microsoft Surface Hub

The Kernel Mode Cryptographic Primitives Library component listed in Section 1.1 was validated using the combination of computers and Windows operating system editions specified in the following table:

Computer	Windows 10 Home	Windows 10 Pro	Windows 10 Enterprise	Windows 10 Education	Surface Hub	Windows 10 S	Windows 10 Mobile
Microsoft Surface Laptop		√	√			√	
Microsoft Surface Pro		√	√	√			
Microsoft Surface Book			√				
Microsoft Surface Pro 4			√				
Microsoft Surface Pro 3		√					
Microsoft Surface 3			√				
Microsoft Surface 3 with LTE		√					
Microsoft Surface Studio			√				
Microsoft Surface Hub					√		
Windows Server 2016 Hyper-V[1]		√					
Microsoft Lumia 950							√
Microsoft Lumia 950 XL							√
Microsoft Lumia 650							√
Dell Latitude 5285		√					
Dell Inspiron 660s	√						
Dell Precision Tower 5810MT		√					
Dell PowerEdge R630		√					
HP Elite X3							√

[1] Host OS: Windows Server 2016, hardware platform: Surface Pro 4

HP Compaq Pro 6305		√					
HP Pro x2 612 G2 Detachable PC with LTE			√				
HP Slimline Desktop		√					
Panasonic Toughbook		√					

1.3 Configure Windows to use FIPS-Approved Cryptographic Algorithms

Use the FIPS Local/Group Security Policy setting or a Mobile Device Management (MDM) to enable FIPS-Approved mode for Kernel Mode Cryptographic Primitives Library.

The Windows operating system provides a group (or local) security policy setting, "System cryptography: Use FIPS compliant algorithms for encryption, hashing, and signing".

Consult the MDM documentation for information on how to enable FIPS-Approved mode. The Policy CSP - Cryptography includes the setting **AllowFipsAlgorithmPolicy**.

Changes to the Approved mode security policy setting do not take effect until the computer has been rebooted.

2 Cryptographic Module Specification

Kernel Mode Cryptographic Primitives Library is a multi-chip standalone module that operates in FIPS-Approved mode during normal operation of the computer and Windows operating system and when Windows is configured to use FIPS-approved cryptographic algorithms as described in Configure Windows to use FIPS-Approved Cryptographic Algorithms.

In addition to configuring Windows to use FIPS-Approved Cryptographic Algorithms, third-party applications and drivers installed on the Windows platform must not use any of the non-approved algorithms implemented by this module. Windows will not operate in an Approved mode when the operators chooses to use a non-Approved algorithm or service

The following configurations and modes of operation will cause Kernel Mode Cryptographic Primitives Library to operate in a non-approved mode of operation:

- Boot Windows in Debug mode
- Boot Windows with Driver Signing disabled

2.1 Cryptographic Boundary

The software cryptographic boundary for Kernel Mode Cryptographic Primitives Library is defined as the binary CNG.SYS.

2.2 FIPS 140-2 Approved Algorithms

Kernel Mode Cryptographic Primitives Library implements the following FIPS-140-2 Approved algorithms:[2]

- FIPS 180-4 SHS SHA-1, SHA-256, SHA-384, and SHA-512 (Cert. # 3790)
- FIPS 198-1 SHA-1, SHA-256, SHA-384, SHA-512 HMAC (Cert. # 3061)
- NIST SP 800-67r1 Triple-DES (2 key legacy-use decryption[3] and 3 key encryption/decryption) in ECB, CBC, CFB8 and CFB64 modes (Cert. # 2459)
- FIPS 197 AES-128, AES-192, and AES-256 in ECB, CBC, CFB8, CFB128, and CTR modes (Cert. # 4624)
- NIST SP 800-38B and SP 800-38C AES-128, AES-192, and AES-256 in CCM and CMAC modes (Cert. # 4624)
- NIST SP 800-38D AES-128, AES-192, and AES-256 GCM decryption and GMAC (Cert. # 4624)
- NIST SP 800-38E XTS-AES XTS-128 and XTS-256 (Cert. # 4624)[4]
- FIPS 186-4 RSA (RSASSA-PKCS1-v1_5 and RSASSA-PSS) digital signature generation and verification with 2048 and 3072 moduli; supporting SHA-1[5], SHA-256, SHA-384, and SHA-512 (Certs. # 2521)
- FIPS 186-4 RSA key-pair generation with 2048 and 3072 moduli (Cert. # 2521)
- FIPS 186-4 ECDSA key pair generation and verification, signature generation and verification with the following NIST curves: P-256, P-384, P-521 (Cert. # 1133)
- FIPS 186-4 DSA PQG generation and verification, signature generation and verification (Cert. # 1223)[6].
- KAS – SP 800-56A Diffie-Hellman Key Agreement; Finite Field Cryptography (FFC) with parameter FB (p=2048, q=224) and FC (p=2048, q=256); key establishment methodology provides 112 bits of encryption strength (Cert. # 127)
- KAS – SP 800-56A EC Diffie-Hellman Key Agreement; Elliptic Curve Cryptography (ECC) with parameter EC (P-256 w/ SHA-256), ED (P-384 w/ SHA-384), and EE (P-521 w/ SHA-512); key establishment methodology provides between 128 and 256-bits of encryption strength (Cert. # 127)
- NIST SP 800-56B RSADP mod 2048 (Cert. # 1281)
- NIST SP 800-90A AES-256 counter mode DRBG (Cert. # 1555)
- NIST SP 800-108 Key Derivation Function (KDF) CMAC-AES (128, 192, 256), HMAC (SHA1, SHA-256, SHA-384, SHA-512) (Cert. # 140)
- NIST SP 800-132 KDF (also known as PBKDF) with HMAC (SHA-1, SHA-256, SHA-384, SHA-512) as the pseudo-random function (vendor affirmed)
- NIST SP 800-38F AES Key Wrapping (128, 192, and 256) (Cert. # 4626)

[2] This module may not use some of the capabilities described in each CAVP certificate.
[3] Two-key Triple-DES Decryption is only allowed for Legacy-usage (as per SP 800-131A). The use of two-key Triple-DES Encryption is disallowed. The caller is responsible for using the key for up to 2^{32} encryptions for IETF protocols and 2^{28} encryptions for any other use.
[4] AES XTS must be used only to protect data at rest and the caller needs to ensure that the length of data encrypted does not exceed 2^{20} AES blocks.
[5] SHA-1 is only acceptable for legacy signature verification.
[6] The DSA functions of signature generation/verification are not supported by this module. DSA functions are not provided as a service, but parts of the DSA algorithm are required as a prerequisite to the KAS FFC implementation contained in this module, which is why DSA is listed here

- NIST SP 800-135 IKEv1 and IKEv2 KDF primitives (Cert. # 1278)[7]
- NIST SP 800-135 TLS primitive (Cert. # 1278)[8]

2.3 Non-Approved Algorithms

Kernel Mode Cryptographic Primitives Library implements the following non-approved algorithms:

- A non-determinic random number generator (NDRNG) that is not a FIPS Approved algorithm but is allowed by FIPS 140. The NDRNG provides entropy input to the DRBG.
- SHA-1 hash, which is disallowed for use in digital signature generation. It can be used for legacy digital signature verification. Its use is Acceptable for non-digital signature generation applications.
- RSA 1024-bits for digital signature generation, which is disallowed.
- NIST SP 800-56A Key Agreement using Finite Field Cryptography (FFC) with parameter FA (p=1024, q=160). The key establishment methodology provides 80 bits of encryption strength instead of the Approved 112 bits of encryption strength listed above.
- If HMAC-SHA1 is used, key sizes less than 112 bits (14 bytes) are not allowed for usage in HMAC generation, as per SP 800-131A.
- MD5 and HMAC-MD5 - allowed for TLS and EAP-TLS
- RC2, RC4, MD2, MD4 (disallowed in FIPS mode)
- 2-Key Triple-DES Encryption, which is disallowed for usage altogether as of the end of 2015.
- DES in ECB, CBC, CFB8 and CFB64 modes (disallowed in FIPS mode)
- Legacy CAPI KDF (proprietary; disallowed in FIPS mode)
- RSA encrypt/decrypt (disallowed in FIPS mode)
- IEEE 1619-2007 XTS-AES, XTS-128 and XTS-256
- NIST SP 800-38D AES-128, AES-192, and AES-256 GCM encryption
- ECDH with the following curves that are allowed in FIPS mode as per FIPS 140-2 IG A.2

Curve	Security Strength (bits)	Allowed in FIPS mode
Curve25519	128	Yes
brainpoolP160r1	80	No
brainpoolP192r1	96	No
brainpoolP192t1	96	No
brainpoolP224r1	112	Yes
brainpoolP224t1	112	Yes
brainpoolP256r1	128	Yes
brainpoolP256t1	128	Yes
brainpoolP320r1	160	Yes
brainpoolP320t1	160	Yes
brainpoolP384r1	192	Yes
brainpoolP384t1	192	Yes
brainpoolP512r1	256	Yes
brainpoolP512t1	256	Yes
ec192wapi	96	No

[7] This cryptographic module supports the IKEv1 and IKEv2 protocols with SP 800-135 rev 1 KDF primitives, however, the protocols have not been reviewed or tested by the NIST CAVP and CMVP.
[8] This cryptographic module supports the TLS protocol with SP 800-135 rev 1 KDF primitive, however, the protocol has not been reviewed or tested by the NIST CAVP and CMVP.

Curve	Security Strength (bits)	Allowed in FIPS mode
nistP192	96	No
nistP224	112	Yes
numsP256t1	128	Yes
numsP384t1	192	Yes
numsP512t1	256	Yes
secP160k1	80	No
secP160r1	80	No
secP160r2	80	No
secP192k1	96	No
secP192r1	96	No
secP224k1	112	Yes
secP224r1	112	Yes
secP256k1	128	Yes
secP256r1	128	Yes
secP384r1	192	Yes
secP521r1	256	Yes
wtls12	112	Yes
wtls7	80	No
wtls9	80	No
x962P192v1	96	No
x962P192v2	96	No
x962P192v3	96	No
x962P239v1	120	Yes
x962P239v2	120	Yes
x962P239v3	120	Yes
x962P256v1	128	Yes

2.4 FIPS 140-2 Approved Algorithms from Bounded Modules

A bounded module is a FIPS 140 module which provides cryptographic functionality that is relied on by a downstream module. As described in the Integrity Chain of Trust section, Kernel Mode Cryptographic Primitives Library depends on the following modules and algorithms:

Implemented in the Windows OS Loader (module certificate #3090):

- CAVP certificate #2523 for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificate #3790 for FIPS 180-4 SHS SHA-256

Implemented in Windows Resume (module certificate #3091):

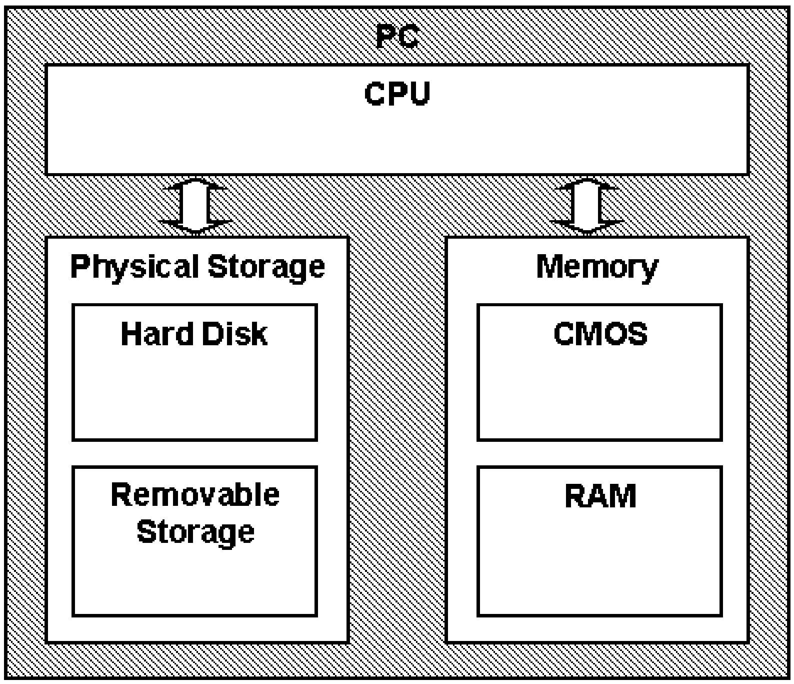
- CAVP certificate #4624 for FIPS 197 AES CBC 128 and 256, SP 800-38E AES XTS 128 and 256

2.5 Cryptographic Bypass

Cryptographic bypass is not supported by Kernel Mode Cryptographic Primitives Library.

2.6 Hardware Components of the Cryptographic Module

The physical boundary of the module is the physical boundary of the computer that contains the module. The following diagram illustrates the hardware components of the Kernel Mode Cryptographic Primitives Library module:



3 Cryptographic Module Ports and Interfaces

The Kernel Mode Cryptographic Primitives Library module implements a set of algorithm providers for the Cryptography Next Generation (CNG) framework in Windows. Each provider in this module represents a single cryptographic algorithm or a set of closely related cryptographic algorithms. These algorithm providers are invoked through the CNG algorithm primitive functions, which are sometimes collectively referred to as the CNG API. For a full list of these algorithm providers, see https://msdn.microsoft.com/en-us/library/aa375534.aspx

The Kernel Mode Cryptographic Primitives Library module is accessed through one of the following logical interfaces:

1. Kernel applications requiring cryptographic services use the BCrypt APIs detailed in Services.
2. Entropy sources supply random bits to the random number generator through the entropy interfaces.

3.1 CNG Primitive Functions

The following security-relevant functions are exported by Kernel Mode Cryptographic Primitives Library:

- BCryptCloseAlgorithmProvider
- BCryptCreateHash
- BCryptCreateMultiHash
- BCryptDecrypt
- BCryptDeriveKey
- BCryptDeriveKeyPBKDF2
- BCryptDestroyHash
- BCryptDestroyKey
- BCryptDestroySecret
- BCryptDuplicateHash
- BCryptDuplicateKey
- BCryptEncrypt
- BCryptExportKey
- BCryptFinalizeKeyPair
- BCryptFinishHash
- BCryptFreeBuffer
- BCryptGenerateKeyPair
- BCryptGenerateSymmetricKey
- BCryptGenRandom
- BCryptGetProperty
- BCryptHash
- BCryptHashData
- BCryptImportKey
- BCryptImportKeyPair
- BCryptKeyDerivation
- BCryptOpenAlgorithmProvider
- BCryptProcessMultiOperations
- BCryptSecretAgreement
- BCryptSetProperty
- BCryptSignHash
- BCryptVerifySignature
- SystemPrng
- EntropyPoolTriggerReseedForIum
- EntropyProvideData
- EntropyRegisterSource
- EntropyUnregisterSource

All of these functions are used in the approved mode. Furthermore, these are the only approved functions that this module can perform.

Kernel Mode Cryptographic Primitives Library has additional export functions described in Non-Security Relevant Configuration Interfaces.

3.1.1 Algorithm Providers and Properties

3.1.1.1 BCryptOpenAlgorithmProvider

```
NTSTATUS WINAPI BCryptOpenAlgorithmProvider(
    BCRYPT_ALG_HANDLE   *phAlgorithm,
    LPCWSTR pszAlgId,
    LPCWSTR pszImplementation,
    ULONG   dwFlags);
```

The BCryptOpenAlgorithmProvider() function has four parameters: algorithm handle output to the opened algorithm provider, desired algorithm ID input, an optional specific provider name input, and optional flags. This function loads and initializes a CNG provider for a given algorithm, and returns a handle to the opened algorithm provider on success.

Unless the calling function specifies the name of the provider, the default provider is used.

The calling function must pass the BCRYPT_ALG_HANDLE_HMAC_FLAG flag in order to use an HMAC function with a hash algorithm.

3.1.1.2 BCryptCloseAlgorithmProvider

```
NTSTATUS WINAPI BCryptCloseAlgorithmProvider(
    BCRYPT_ALG_HANDLE   hAlgorithm,
    ULONG   dwFlags);
```

This function closes an algorithm provider handle opened by a call to BCryptOpenAlgorithmProvider() function.

3.1.1.3 BCryptSetProperty

```
NTSTATUS WINAPI BCryptSetProperty(
    BCRYPT_HANDLE   hObject,
    LPCWSTR pszProperty,
    PUCHAR   pbInput,
    ULONG   cbInput,
    ULONG   dwFlags);
```

The BCryptSetProperty() function sets the value of a named property for a CNG object. The CNG object is a handle, the property name is a NULL terminated string, and the value of the property is a length-specified byte string.

3.1.1.4 BCryptGetProperty

```
NTSTATUS WINAPI BCryptGetProperty(
    BCRYPT_HANDLE   hObject,
    LPCWSTR pszProperty,
    PUCHAR   pbOutput,
    ULONG   cbOutput,
    ULONG   *pcbResult,
    ULONG   dwFlags);
```

The BCryptGetProperty() function retrieves the value of a named property for a CNG object. The CNG object is a handle, the property name is a NULL terminated string, and the value of the property is a length-specified byte string.

3.1.1.5 BCryptFreeBuffer

```
VOID WINAPI BCryptFreeBuffer(
        PVOID   pvBuffer);
```

Some of the CNG functions allocate memory on caller's behalf. The BCryptFreeBuffer() function frees memory that was allocated by such a CNG function.

3.1.2 Random Number Generation

3.1.2.1 BCryptGenRandom

```
NTSTATUS WINAPI BCryptGenRandom(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        PUCHAR  pbBuffer,
        ULONG   cbBuffer,
        ULONG   dwFlags);
```

The BCryptGenRandom() function fills a buffer with random bytes. The random number generation algorithm is:

- BCRYPT_RNG_ALGORITHM. This is the AES-256 counter mode based random generator as defined in SP 800-90A.

This function is a wrapper for SystemPrng.

3.1.2.2 SystemPrng

```
BOOL SystemPrng(
        unsigned char   *pbRandomData,
        size_t          cbRandomData );
```

The SystemPrng() function fills a buffer with random bytes generated from output of NIST SP 800-90A AES-256 counter mode based DRBG seeded from the Windows entropy pool. The Windows entropy pool is populated from the following sources:

- An initial entropy value provided by the Windows OS Loader at boot time.
- The values of the high-resolution CPU cycle counter at times when hardware interrupts are received.
- Random values gathered from the Trusted Platform Module (TPM), if one is available on the system.
- Random values gathered by calling the RDRAND CPU instruction, if supported by the CPU.

The Windows DRBG infrastructure located in cng.sys continues to gather entropy from these sources during normal operation, and the DRBG cascade is periodically reseeded with new entropy.

3.1.2.3 EntropyRegisterSource

```
NTSTATUS EntropyRegisterSource(
        ENTROPY_SOURCE_HANDLE * phEntropySource,
        ENTROPY_SOURCE_TYPE     entropySourceType,
    PCWSTR            entropySourceName );
```

This function is used to obtain a handle that can be used to contribute randomness to the Windows entropy pool. The handle is returned in the phEntropySource parameter. For this function, entropySource must be set to ENTROPY_SOURCE_TYPE_HIGH_PUSH, and entropySourceName must be a Unicode string describing the entropy source.

3.1.2.4 EntropyUnregisterSource

```
NTSTATUS EntropyRegisterSource(
    ENTROPY_SOURCE_HANDLE hEntropySource);
```

This function is used to destroy a handle created with EntropyRegisterSource().

3.1.2.5 EntropyProvideData

```
NTSTATUS EntropyProvideData(
        ENTROPY_SOURCE_HANDLE   hEntropySource,
        PCBYTE            pbData,
        SIZE_T            cbData,
        ULONG             entropyEstimateInMilliBits );
```

This function is used to contribute entropy to the Windows entropy pool. hEntropySource must be a handle returned by an earlier call to EntropyRegisterSource. The caller provides cbData bytes in the buffer pointed to by pbData, as well as an estimate (in the entropyEstimateInMilliBits parameter) of how many millibits of entropy are contained in these bytes.

3.1.2.6 EntropyPoolTriggerReseedForIum

```
VOID EntropyPoolTriggerReseedForIum(BOOLEAN fPerformCallbacks);
```

This function will trigger a kernel DRBG reseed for the cng.sys inside the IUM (Isolated User Mode) environment. If called inside the IUM environment, it triggers a reseed from one or more of the entropy pools of the system. If called inside the normal world (non-IUM) environment, this function does nothing.

3.1.3 Key and Key-Pair Generation

3.1.3.1 BCryptGenerateSymmetricKey

```
NTSTATUS WINAPI BCryptGenerateSymmetricKey(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        BCRYPT_KEY_HANDLE   *phKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        PUCHAR   pbSecret,
        ULONG   cbSecret,
```

```
        ULONG   dwFlags);
```

The BCryptGenerateSymmetricKey() function generates a symmetric key object directly from a DRBG for use with a symmetric encryption algorithm or key derivation algorithm from a supplied key value. The calling application must specify a handle to the algorithm provider created with the BCryptOpenAlgorithmProvider() function. The algorithm specified when the provider was created must support symmetric key encryption or key derivation.

3.1.3.2 BCryptGenerateKeyPair

```
    NTSTATUS WINAPI BCryptGenerateKeyPair(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        BCRYPT_KEY_HANDLE   *phKey,
        ULONG   dwLength,
        ULONG   dwFlags);
```

The BCryptGenerateKeyPair() function creates an empty public/private key pair. After creating a key using this function, call the BCryptSetProperty() function to set its properties. The key pair can be used only after BCryptFinalizeKeyPair() function is called.

3.1.3.3 BCryptFinalizeKeyPair

```
    NTSTATUS WINAPI BCryptFinalizeKeyPair(
        BCRYPT_KEY_HANDLE   hKey,
        ULONG   dwFlags);
```

The BCryptFinalizeKeyPair() function completes a public/private key pair import or generation directly from the output of a DRBG. The key pair cannot be used until this function has been called. After this function has been called, the BCryptSetProperty() function can no longer be used for this key.

3.1.3.4 BCryptDuplicateKey

```
    NTSTATUS WINAPI BCryptDuplicateKey(
        BCRYPT_KEY_HANDLE   hKey,
        BCRYPT_KEY_HANDLE   *phNewKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        ULONG   dwFlags);
```

The BCryptDuplicateKey() function creates a duplicate of a symmetric key.

3.1.3.5 BCryptDestroyKey

```
    NTSTATUS WINAPI BCryptDestroyKey(
        BCRYPT_KEY_HANDLE   hKey);
```

The BCryptDestroyKey() function destroys the specified key.

3.1.4 Key Entry and Output

3.1.4.1 BCryptImportKey

```
    NTSTATUS WINAPI BCryptImportKey(
        BCRYPT_ALG_HANDLE hAlgorithm,
```

```
        BCRYPT_KEY_HANDLE hImportKey,
        LPCWSTR pszBlobType,
        BCRYPT_KEY_HANDLE *phKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptImportKey() function imports a symmetric key from a key blob.

3.1.4.2 BCryptImportKeyPair

```
NTSTATUS WINAPI BCryptImportKeyPair(
        BCRYPT_ALG_HANDLE hAlgorithm,
        BCRYPT_KEY_HANDLE hImportKey,
        LPCWSTR pszBlobType,
        BCRYPT_KEY_HANDLE *phKey,
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptImportKeyPair() function is used to import a public/private key pair from a key blob.

3.1.4.3 BCryptExportKey

```
NTSTATUS WINAPI BCryptExportKey(
        BCRYPT_KEY_HANDLE   hKey,
        BCRYPT_KEY_HANDLE   hExportKey,
        LPCWSTR pszBlobType,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptExportKey() function exports a key to a memory blob that can be persisted for later use.

3.1.5 Encryption and Decryption

3.1.5.1 BCryptEncrypt

```
NTSTATUS WINAPI BCryptEncrypt(
        BCRYPT_KEY_HANDLE hKey,
        PUCHAR   pbInput,
        ULONG   cbInput,
        VOID    *pPaddingInfo,
        PUCHAR   pbIV,
```



```
        ULONG   cbIV,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptEncrypt() function encrypts a block of data of given length.

3.1.5.2 BCryptDecrypt

```
    NTSTATUS WINAPI BCryptDecrypt(
        BCRYPT_KEY_HANDLE   hKey,
        PUCHAR   pbInput,
        ULONG   cbInput,
        VOID    *pPaddingInfo,
        PUCHAR   pbIV,
        ULONG   cbIV,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptDecrypt() function decrypts a block of data of given length.

3.1.6 Hashing and Message Authentication

3.1.6.1 BCryptCreateHash

```
    NTSTATUS WINAPI BCryptCreateHash(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        BCRYPT_HASH_HANDLE  *phHash,
        PUCHAR   pbHashObject,
        ULONG   cbHashObject,
        PUCHAR   pbSecret,
        ULONG   cbSecret,
        ULONG   dwFlags);
```

The BCryptCreateHash() function creates a hash object with an optional key. The optional key is used for HMAC, AES GMAC and AES CMAC.

3.1.6.2 BCryptHashData

```
    NTSTATUS WINAPI BCryptHashData(
        BCRYPT_HASH_HANDLE  hHash,
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptHashData() function performs a one way hash on a data buffer. Call the BCryptFinishHash() function to finalize the hashing operation to get the hash result.

3.1.6.3 BCryptDuplicateHash

```
NTSTATUS WINAPI BCryptDuplicateHash(
        BCRYPT_HASH_HANDLE  hHash,
        BCRYPT_HASH_HANDLE  *phNewHash,
        PUCHAR   pbHashObject,
        ULONG   cbHashObject,
        ULONG   dwFlags);
```

The BCryptDuplicateHash()function duplicates an existing hash object. The duplicate hash object contains all state and data that was hashed to the point of duplication.

3.1.6.4 BCryptFinishHash

```
NTSTATUS WINAPI BCryptFinishHash(
        BCRYPT_HASH_HANDLE hHash,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   dwFlags);
```

The BCryptFinishHash() function retrieves the hash value for the data accumulated from prior calls to BCryptHashData() function.

3.1.6.5 BCryptDestroyHash

```
NTSTATUS WINAPI BCryptDestroyHash(
        BCRYPT_HASH_HANDLE  hHash);
```

The BCryptDestroyHash() function destroys a hash object.

3.1.6.6 BCryptHash

```
NTSTATUS WINAPI BCryptHash(
        BCRYPT_ALG_HANDLE hAlgorithm,
        PUCHAR pbSecret,
        ULONG cbSecret,
        PUCHAR pbInput,
        ULONG cbInput,
        PUCHAR pbOutput,
        ULONG  cbOutput);
```

The function BCryptHash() performs a single hash computation. This is a convenience function that wraps calls to the BCryptCreateHash(), BCryptHashData(), BCryptFinishHash(), and BCryptDestroyHash() functions.

3.1.6.7 BCryptCreateMultiHash

```
NTSTATUS WINAPI BCryptCreateMultiHash(
        BCRYPT_ALG_HANDLE hAlgorithm,
        BCRYPT_HASH_HANDLE *phHash,
```

```
        ULONG nHashes,
        PUCHAR pbHashObject,
        ULONG cbHashObject,
        PUCHAR pbSecret,
        ULONG cbSecret,
        ULONG dwFlags);
```

BCryptCreateMultiHash() is a function that creates a new MultiHash object that is used in parallel hashing to improve performance. The MultiHash object is equivalent to an array of normal (reusable) hash objects.

3.1.6.8 BCryptProcessMultiOperations

```
NTSTATUS WINAPI BCryptProcessMultiOperations(
        BCRYPT_HANDLE  hObject,
        BCRYPT_MULTI_OPERATION_TYPE operationType,
        PVOID pOperations,
        ULONG cbOperations,
        ULONG dwFlags );
```

The BCryptProcessMultiOperations() function is used to perform multiple operations on a single multi-object handle such as a MultiHash object handle. If any of the operations fail, then the function will return an error.

Each element of the operations array specifies an operation to be performed on/with the hObject.

For hash operations, there are two operation types:

- Hash data
- Finalize hash

These correspond directly to BCryptHashData() and BCryptFinishHash(). Each operation specifies an index of the hash object inside the hObject MultiHash object that this operation applies to. Operations are executed in any order or even in parallel, with the sole restriction that the set of operations that specify the same index are all executed in-order.

3.1.7 Signing and Verification

3.1.7.1 BCryptSignHash

```
NTSTATUS WINAPI BCryptSignHash(
        BCRYPT_KEY_HANDLE   hKey,
        VOID    *pPaddingInfo,
        PUCHAR   pbInput,
        ULONG   cbInput,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
```



```
        ULONG   dwFlags);
```

The BCryptSignHash() function creates a signature of a hash value.
Note: this function accepts SHA-1 hashes, which according to NIST SP 800-131A is *disallowed* for digital signature generation. SHA-1 is currently *legacy-use* for digital signature verification.

3.1.7.2 BCryptVerifySignature

```
    NTSTATUS WINAPI BCryptVerifySignature(
        BCRYPT_KEY_HANDLE   hKey,
        VOID    *pPaddingInfo,
        PUCHAR   pbHash,
        ULONG   cbHash,
        PUCHAR   pbSignature,
        ULONG   cbSignature,
        ULONG   dwFlags);
```

The BCryptVerifySignature() function verifies that the specified signature matches the specified hash.
Note: this function accepts SHA-1 hashes, which according to NIST SP 800-131A is *disallowed* for digital signature generation. SHA-1 is currently *legacy-use* for digital signature verification.

3.1.8 Secret Agreement and Key Derivation

3.1.8.1 BCryptSecretAgreement

```
    NTSTATUS WINAPI BCryptSecretAgreement(
        BCRYPT_KEY_HANDLE      hPrivKey,
        BCRYPT_KEY_HANDLE      hPubKey,
        BCRYPT_SECRET_HANDLE   *phAgreedSecret,
        ULONG                  dwFlags);
```

The BCryptSecretAgreement() function creates a secret agreement value from a private and a public key. This function is used with Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH) algorithms.

3.1.8.2 BCryptDeriveKey

```
    NTSTATUS WINAPI BCryptDeriveKey(
        BCRYPT_SECRET_HANDLE hSharedSecret,
        LPCWSTR          pwszKDF,
        BCryptBufferDesc     *pParameterList,
        PUCHAR pbDerivedKey,
        ULONG           cbDerivedKey,
        ULONG           *pcbResult,
        ULONG           dwFlags);
```

The BCryptDeriveKey() function derives a key from a secret agreement value.
Note: When supporting a key agreement scheme that requires a nonce, BCryptDeriveKey uses whichever nonce is supplied by the caller in the BCryptBufferDesc. Examples of the nonce types are found in Section 5.4 of http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf

When using a nonce, a random nonce **should** be used. And then if the random nonce is used, the entropy (amount of randomness) of the nonce and the security strength of the DRBG has to be at least one half of the minimum required bit length of the subgroup order.

For example:

for KAS FFC, entropy of nonce must be 112 bits for FB, 128 bits for FC.

for KAS ECC, entropy of the nonce must be 128 bits for EC, 182 for ED, 256 for EF.

3.1.8.3 BCryptDestroySecret

```
NTSTATUS WINAPI BCryptDestroySecret(
        BCRYPT_SECRET_HANDLE   hSecret);
```

The BCryptDestroySecret() function destroys a secret agreement handle that was created by using the BCryptSecretAgreement() function.

3.1.8.4 BCryptKeyDerivation

```
NTSTATUS WINAPI BCryptKeyDerivation(
        _In_       BCRYPT_KEY_HANDLE hKey,
        _In_opt_   BCryptBufferDesc     *pParameterList,
        _Out_writes_bytes_to_(cbDerivedKey, *pcbResult) PUCHAR pbDerivedKey,
        _In_       ULONG            cbDerivedKey,
        _Out_      ULONG            *pcbResult,
        _In_       ULONG            dwFlags);
```

The BCryptKeyDerivation() function executes a Key Derivation Function (KDF) on a key generated with BCryptGenerateSymmetricKey() function. It differs from the BCryptDeriveKey() function in that it does not require a secret agreement step to create a shared secret.

3.1.8.5 BCryptDeriveKeyPBKDF2

```
NTSTATUS WINAPI BCryptDeriveKeyPBKDF2(
        BCRYPT_ALG_HANDLE hPrf,
        PUCHAR pbPassword,
        ULONG cbPassword,
        PUCHAR pbSalt,
        ULONG cbSalt,
        ULONGLONG cIterations,
        PUCHAR pbDerivedKey,
        ULONG cbDerivedKey,
        ULONG dwFlags);
```

The BCryptDeriveKeyPBKDF2() function derives a key from a hash value by using the password based key derivation function as defined by SP 800-132 PBKDF and IETF RFC 2898 (specified as PBKDF2).

3.1.9 Cryptographic Transitions

3.1.9.1 DH and ECDH

Through the year 2010, implementations of DH and ECDH were allowed to have an acceptable bit strength of at least 80 bits of security (for DH at least 1024 bits and for ECDH at least 160 bits). From 2011 through 2013, 80 bits of security strength was considered deprecated, and was disallowed starting January 1, 2014. As of that date, only security strength of at least 112 bits is acceptable. ECDH uses curve sizes of at least 256 bits (that means it has at least 128 bits of security strength), so that is acceptable. However, DH has a range of 1024 to 4096 and that changed to 2048 to 4096 after 2013.

3.1.9.2 SHA-1

From 2011 through 2013, SHA-1 could be used in a deprecated mode for use in digital signature generation. As of Jan. 1, 2014, SHA-1 is no longer allowed for digital signature generation, and it is allowed for legacy use only for digital signature verification.

3.2 Control Input Interface

The Control Input Interface are the functions in Algorithm Providers and Properties. Options for control operations are passed as input parameters to these functions.

3.3 Status Output Interface

The Status Output Interface for Kernel Mode Cryptographic Primitives Library is the return value from each export function in the Kernel Mode Cryptographic Primitives Library.

3.4 Data Output Interface

The Data Output Interface for Kernel Mode Cryptographic Primitives Library consists of the Kernel Mode Cryptographic Primitives Library export functions except for the Control Input Interfaces. Data is returned to the function's caller via output parameters.

3.5 Data Input Interface

The Data Input Interface for Kernel Mode Cryptographic Primitives Library consists of the Kernel Mode Cryptographic Primitives Library export functions except for the Control Input Interfaces. Data and options are passed to the interface as input parameters to the export functions. Data Input is kept separate from Control Input by passing Data Input in separate parameters from Control Input.

3.6 Non-Security Relevant Configuration Interfaces

The following interfaces are not cryptographic functions and are used to configure cryptographic providers on the system. Please see https://msdn.microsoft.com for details.

Function Name	Description
BCryptEnumAlgorithms	Enumerates the algorithms for a given set of operations.
BCryptEnumProviders	Returns a list of CNG providers for a given algorithm.
BCryptRegisterConfigChangeNotify	This is deprecated beginning with Windows 10.

BCryptResolveProviders	Resolves queries against the set of providers currently registered on the local system and the configuration information specified in the machine and domain configuration tables, returning an ordered list of references to one or more providers matching the specified criteria.
BCryptAddContextFunctionProvider	Adds a cryptographic function provider to the list of providers that are supported by an existing CNG context.
BCryptRegisterProvider	Registers a CNG provider.
BCryptUnregisterProvider	Unregisters a CNG provider.
BCryptUnregisterConfigChangeNotify	Removes a CNG configuration change event handler. This API differs slightly between User-Mode and Kernel-Mode.
BCryptGetFipsAlgorithmMode **CngGetFipsAlgorithmMode**	Determines whether Kernel Mode Cryptographic Primitives Library is operating in FIPS mode. Some applications use the value returned by this API to alter their own behavior, such as blocking the use of some SSL versions.
EntropyRegisterCallback	Registers the callback function that will be called in a worker thread after every reseed that the system performs. The callback is merely informational.
BCryptQueryProviderRegistration	Retrieves information about a CNG provider.
BCryptEnumRegisteredProviders	Retrieves information about the registered providers.
BCryptCreateContext	Creates a new CNG configuration context.
BCryptDeleteContext	Deletes an existing CNG configuration context.
BCryptEnumContexts	Obtains the identifiers of the contexts in the specified configuration table.
BCryptConfigureContext	Sets the configuration information for an existing CNG context.
BCryptQueryContextConfiguration	Retrieves the current configuration for the specified CNG context.
BCryptAddContextFunction	Adds a cryptographic function to the list of functions that are supported by an existing CNG context.
BCryptRemoveContextFunction	Removes a cryptographic function from the list of functions that are supported by an existing CNG context.
BCryptEnumContextFunctions	Obtains the cryptographic functions for a context in the specified configuration table.
BCryptConfigureContextFunction	Sets the configuration information for the cryptographic function of an existing CNG context.
BCryptQueryContextFunctionConfiguration	Obtains the cryptographic function configuration information for an existing CNG context.
BCryptEnumContextFunctionProviders	Obtains the providers for the cryptographic functions for a context in the specified configuration table.
BCryptSetContextFunctionProperty	Sets the value of a named property or a cryptographic

	function in an existing CNG context.
BCryptQueryContextFunctionProperty	Obtains the value of a named property for a cryptographic function in an existing CNG context.
BCryptSetAuditingInterface	Sets the auditing interface.

4 Roles, Services and Authentication

4.1 Roles

Kernel Mode Cryptographic Primitives Library is a kernel-mode driver that does not interact with the user through any service therefore the module's functions are fully automatic and not configurable. FIPS 140 validations define formal "User" and "Cryptographic Officer" roles. Both roles can use any of this module's services.

4.2 Services

Kernel Mode Cryptographic Primitives Library services are:

1. **Algorithm Providers and Properties** – This module provides interfaces to register algorithm providers
2. **Random Number Generation**
3. **Key and Key-Pair Generation**
4. **Key Entry and Output**
5. **Encryption and Decryption**
6. **Hashing and Message Authentication**
7. **Signing and Verification**
8. **Secret Agreement and Key Derivation**
9. **Show Status**
10. **Self-Tests** - The module provides a power-up self-tests service that is automatically executed when the module is loaded into memory. See Self-Tests.
11. **Zeroizing Cryptographic Material -** See Cryptographic Key Management

4.2.1 Mapping of Services, Algorithms, and Critical Security Parameters

The following table maps the services to their corresponding algorithms and critical security parameters (CSPs).

Service	Algorithms	CSPs
Algorithm Providers and Properties	None	None
Random Number Generation	AES-256 CTR DRBG NDRNG (allowed, used to provide entropy to DRBG)	AES-CTR DRBG Entropy Input AES-CTR DRBG Seed AES-CTR DRBG V AES-CTR DRBG Key
Key and Key-Pair Generation	RSA, DH, ECDH, ECDSA, RC2, RC4,	Symmetric Keys

	DES, Triple-DES, AES, and HMAC (RC2, RC4, and DES cannot be used in FIPS mode.)	Asymmetric Public Keys Asymmetric Private Keys
Key Entry and Output	SP 800-38F AES Key Wrapping (128, 192, and 256)	Symmetric Keys Asymmetric Public Keys Asymmetric Private Keys
Encryption and Decryption	• Triple-DES with 2 key (encryption disallowed) and 3 key in ECB, CBC, CFB8 and CFB64 modes; • AES-128, AES-192, and AES-256 in ECB, CBC, CFB8, CFB128, and CTR modes; • AES-128, AES-192, and AES-256 in CCM, CMAC, and GMAC modes; • AES-128, AES-192, and AES-256 GCM decryption; • NIST SP XTS-AES XTS-128 and XTS-256; • SP 800-56B RSADP mod 2048 (IEEE 1619-2007 XTS-AES, AES GCM encryption, RC2, RC4, RSA, and DES, which cannot be used in FIPS mode)	Symmetric Keys Asymmetric Public Keys Asymmetric Private Keys
Hashing and Message Authentication	• FIPS 180-4 SHS SHA-1, SHA-256, SHA-384, and SHA-512; • FIPS 180-4 SHA-1, SHA-256, SHA-384, SHA-512 HMAC; • AES-128, AES-192, and AES-256 in CCM, CMAC, and GMAC; • MD5 and HMAC-MD5 (allowed in TLS and EAP-TLS); • MD2 and MD4 (disallowed in FIPS mode)	Symmetric Keys (for HMAC, AES CCM, AES CMAC, and AES GMAC)
Signing and Verification	• FIPS 186-4 RSA (RSASSA-PKCS1-v1_5 and RSASSA-PSS) digital signature generation and verification with 2048 and 3072 modulus; supporting SHA-1[9], SHA-256, SHA-384, and SHA-512	Asymmetric Public Keys Asymmetric RSA Private Keys Asymmetric ECDSA Public Keys Asymmetric ECDSA Private keys

[9] SHA-1 is only acceptable for legacy signature verification.

	• FIPS 186-4 ECDSA with the following NIST curves: P-256, P-384, P-521	
Secret Agreement and Key Derivation	• KAS – SP 800-56A Diffie-Hellman Key Agreement; Finite Field Cryptography (FFC) • KAS – SP 800-56A EC Diffie-Hellman Key Agreement with the following NIST curves: P-256, P-384, P-521 and the FIPS non-Approved curves listed in Non-Approved Algorithms • SP 800-108 Key Derivation Function (KDF) CMAC-AES (128, 192, 256), HMAC (SHA1, SHA-256, SHA-384, SHA-512) • SP 800-132 PBKDF • SP 800-135 IKEv1 and IKEv2 KDF primitives • Legacy CAPI KDF (cannot be used in FIPS mode)	DH Private and Public Values ECDH Private and Public Values
Show Status	None	None
Self-Tests	See Section 8 Self-Tests for the list of algorithms	None
Zeroizing Cryptographic Material	None	None

4.2.2 Mapping of Services, Export Functions, and Invocations

The following table maps the services to their corresponding export functions and invocations.

Service	Export Functions	Invocations
Algorithm Providers and Properties	BCryptOpenAlgorithmProvider BCryptCloseAlgorithmProvider BCryptSetProperty BCryptGetProperty BCryptFreeBuffer	This service is executed whenever one of these exported functions is called.
Random Number Generation	BcryptGenRandom SystemPrng EntropyRegisterSource EntropyUnregisterSource EntropyProvideData EntropyPoolTriggerReseedForIum	This service is executed whenever one of these exported functions is called.
Key and Key-Pair Generation	BCryptGenerateSymmetricKey	This service is executed

	BCryptGenerateKeyPair BCryptFinalizeKeyPair BCryptDuplicateKey BCryptDestroyKey	whenever one of these exported functions is called.
Key Entry and Output	BCryptImportKey BCryptImportKeyPair BCryptExportKey	This service is executed whenever one of these exported functions is called.
Encryption and Decryption	BCryptEncrypt BCryptDecrypt	This service is executed whenever one of these exported functions is called.
Hashing and Message Authentication	BCryptCreateHash BCryptHashData BCryptDuplicateHash BCryptFinishHash BCryptDestroyHash BCryptHash BCryptCreateMultiHash BCryptProcessMultiOperations	This service is executed whenever one of these exported functions is called.
Signing and Verification	BCryptSignHash BCryptVerifySignature	This service is executed whenever one of these exported functions is called.
Secret Agreement and Key Derivation	BCryptSecretAgreement BCryptDeriveKey BCryptDestroySecret BCryptKeyDerivation BCryptDeriveKeyPBKDF2	This service is executed whenever one of these exported functions is called.
Show Status	All Exported Functions	This service is executed upon completion of an exported function.
Self-Tests	DriverEntry	This service is executed upon startup of this module.
Zeroizing Cryptographic Material	BCryptDestroyKey BCryptDestroySecret	This service is executed whenever one of these exported functions is called.

4.2.3 Non-Approved Services

The following table lists other non-security relevant or non-approved APIs exported from the crypto module.

Function Name	Description
BCryptDeriveKeyCapi	Derives a key from a hash value. This function is provided as a helper function to assist in migrating from legacy Cryptography API (CAPI) to CNG.
SslDecryptPacket **SslEncryptPacket**	Supports Secure Sockets Layer (SSL) protocol functionality. These functions are non-approved.

SslExportKey **SslFreeObject** **SslImportKey** **SslLookupCipherLengths** **SslLookupCipherSuiteInfo** **SslOpenProvider** **SslIncrementProviderReferenceCount** **SslDecrementProviderReferenceCount**	

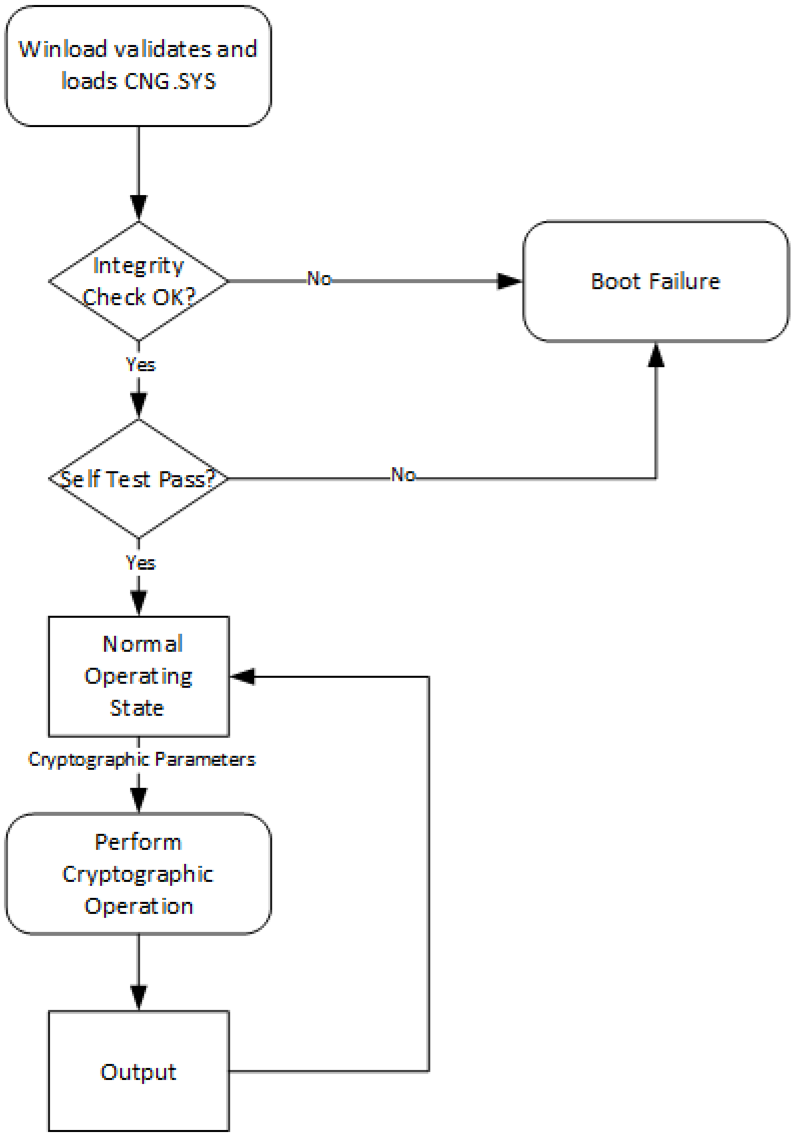
4.3 Authentication

The module does not provide authentication. Roles are implicitly assumed based on the services that are executed.

5 Finite State Model

5.1 Specification

The following diagram shows the finite state model for Kernel Mode Cryptographic Primitives Library:



6 Operational Environment

The operational environment for Kernel Mode Cryptographic Primitives Library is the Windows 10 operating system running on a supported hardware platform.

6.1 Single Operator

The for Kernel Mode Cryptographic Primitives Library is loaded into kernel memory as part of the boot process and before the logon component is initialized. The "single operator" for the module is the Windows Kernel.

6.2 Cryptographic Isolation

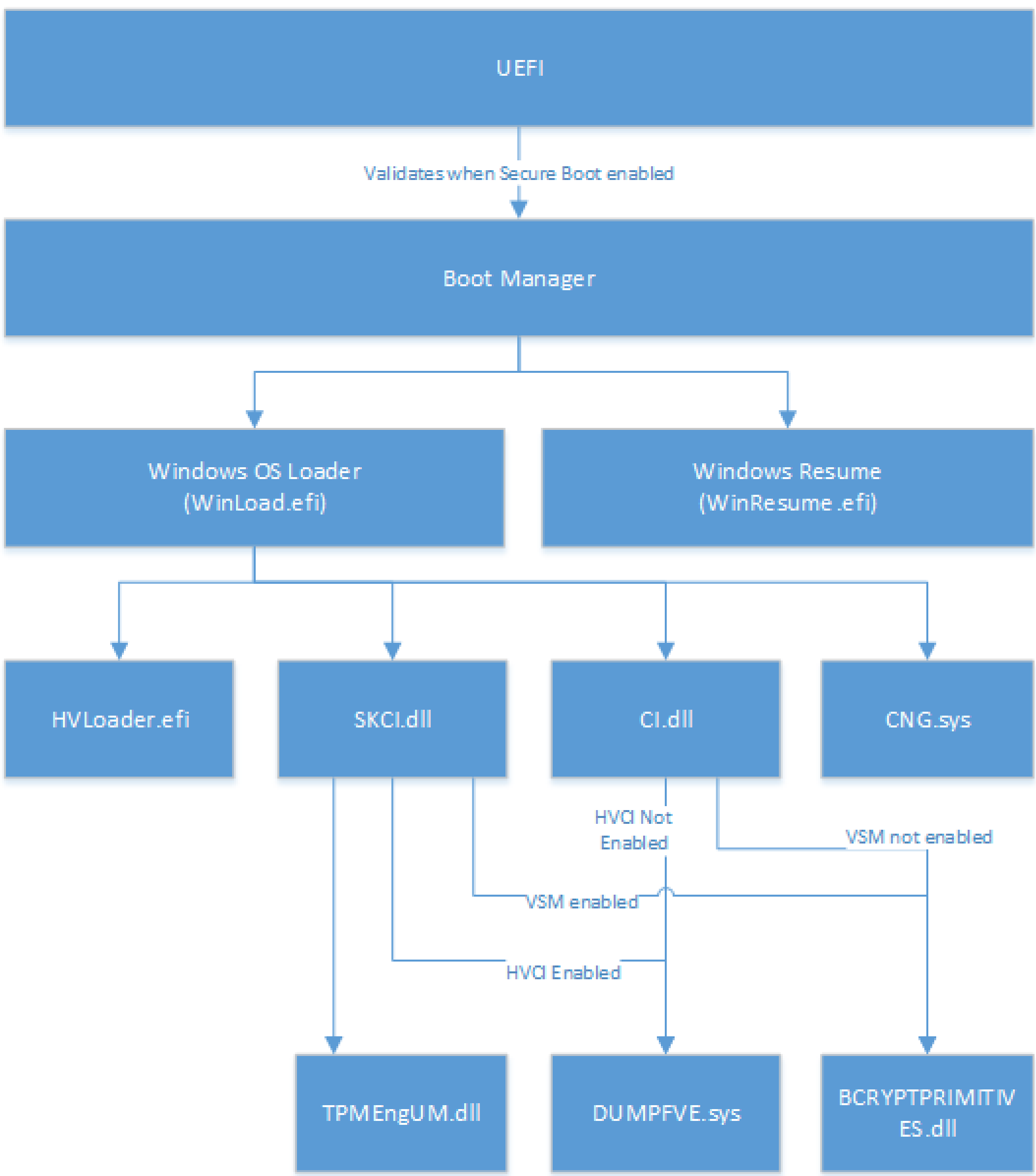
In the Windows operating system, all kernel-mode modules, including CNG.SYS, are loaded into the Windows Kernel (ntoskrnl.exe) which executes as a single process. The Windows operating system

environment enforces process isolation from user-mode processes including memory and CPU scheduling between the kernel and user-mode processes.

6.3 Integrity Chain of Trust

Modules running in the Windows OS environment provide integrity verification through different mechanisms depending on when the module loads in the OS load sequence and also on the hardware and OS configuration. The following diagrams describe the Integrity Chain of trust for each supported configuration that impacts integrity checks:

Boot Sequence and Chain of Trust



The Windows OS Loader checks the integrity of Kernel Mode Cryptographic Primitives Library before it is loaded into ntoskrnl.exe.

Windows binaries include a SHA-256 hash of the binary signed with the 2048 bit Microsoft RSA code-signing key (i.e., the key associated with the Microsoft code-signing certificate). The integrity check uses the public key component of the Microsoft code signing certificate to verify the signed hash of the binary.

7 Cryptographic Key Management

The Kernel Mode Cryptographic Primitives Library crypto module uses the following critical security parameters (CSPs) for FIPS Approved security functions:

Security Relevant Data Item	Description
Symmetric encryption/decryption keys	Keys used for AES or Triple-DES encryption/decryption. Key sizes for AES are 128, 192, and 256 bits, and key sizes for Triple-DES are 192 and 128 bits.
HMAC keys	Keys used for HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, and HMAC-SHA512
Asymmetric ECDSA Public Keys	Keys used for the verification of ECDSA digital signatures. Curve sizes are P-256, P-384, and P-521.
Asymmetric ECDSA Private Keys	Keys used for the calculation of ECDSA digital signatures. Curve sizes are P-256, P-384, and P-521.
Asymmetric RSA Public Keys	Keys used for the verification of RSA digital signatures. Key sizes are 2048 and 3072 bits. These keys can be produced using RSA Key Generation.
Asymmetric RSA Private Keys	Keys used for the calculation of RSA digital signatures. Key sizes are 2048 and 3072 bits. These keys can be produced using RSA Key Generation.
AES-CTR DRBG Entropy Input	A secret value that is at least 256 bits and maintained internal to the module that provides the entropy material for AES-CTR DRBG output[10]
AES-CTR DRBG Seed	A 384 bit secret value maintained internal to the module that provides the seed material for AES-CTR DRBG output[11]
AES-CTR DRBG V	A 128 bit secret value maintained internal to the module

[10] Microsoft Common Criteria Windows Security Target, Page 29.
[11] Recommendation for Random Number Generation Using Deterministic Random Bit Generators, NIST SP 800-90A Revision 1, page 49.

	that provides the entropy material for AES-CTR DRBG output[12]
AES-CTR DRBG Key	A 256 bit secret value maintained internal to the module that provides the entropy material for AES-CTR DRBG output[13]
DH Private and Public values	Private and public values used for Diffie-Hellman key establishment. Key sizes are 2048 to 4096 bits.
ECDH Private and Public values	Private and public values used for EC Diffie-Hellman key establishment. Curve sizes are P-256, P-384, and P-521 and the ones listed in section 2.3.

7.1 Access Control Policy

The Kernel Mode Cryptographic Primitives Library crypto module allows controlled access to the security relevant data items contained within it. The following table defines the access that a service has to each. The permissions are categorized as a set of four separate permissions: read (r), write (w), execute (x), delete (d). If no permission is listed, the service has no access to the item.

Kernel Mode Cryptographic Primitives Library crypto module **Service Access Policy**	Symmetric encryption/decryption keys	HMAC keys	Asymmetric ECDSA Public keys	Asymmetric ECDSA Private keys	Asymmetric RSA Public Keys	Asymmetric RSA Private Keys	DH Public and Private values	ECDH Public and Private values	AES-CTR DRBG Seed, AES-CTR DRBG Entropy Input, AES-CTR DRBG V, & AES-CTR DRBG key
Algorithm Providers and Properties									
Random Number Generation									x
Key and Key-Pair Generation	wd	wd	wd	wd	wd	wd	wd	wd	x
Key Entry and Output	rw	rw	rw	rw	rw	rw	rw	rw	
Encryption and Decryption	x								
Hashing and Message Authentication		wx							
Signing and Verification			x	x	x	x			x
Secret Agreement and Key Derivation							x	x	x
Show Status									

[12] Ibid.
[13] Ibid.

Self-Tests									
Zeroizing Cryptographic Material	wd	wd	wd	wd	wd	wd	wd	wd	wd

7.2 Key Material

When Kernel Mode Cryptographic Primitives Library is loaded in the Windows 10 operating system kernel, no keys exist within it. A kernel module is responsible for importing keys into Kernel Mode Cryptographic Primitives Library or using Kernel Mode Cryptographic Primitives Library's functions to generate keys.

7.3 Key Generation

Kernel Mode Cryptographic Primitives Library can create and use keys for the following algorithms: RSA, DH, ECDH, ECDSA, RC2, RC4, DES, Triple-DES, AES, and HMAC. However, RC2, RC4, and DES cannot be used in FIPS mode.

Random keys can be generated by calling the BCryptGenerateSymmetricKey() and BCryptGenerateKeyPair() functions. Random data generated by the BCryptGenRandom() function is provided to BCryptGenerateSymmetricKey() function to generate symmetric keys. DES, Triple-DES, AES, RSA, ECDSA, DH, and ECDH keys and key-pairs are generated following the techniques given in SP 800-56Ar2 (Section 5.8.1).

Keys generated while not operating in the FIPS mode of operation cannot be used in FIPS mode, and vice versa.

7.4 Key Establishment

Kernel Mode Cryptographic Primitives Library can use FIPS approved Diffie-Hellman key agreement (DH), Elliptic Curve Diffie-Hellman key agreement (ECDH), RSA key transport and manual methods to establish keys. Alternatively, the module can also use Approved KDFs to derive key material from a specified secret value or password.

Kernel Mode Cryptographic Primitives Library can use the following FIPS Approved key derivation functions (KDF) from the common secret that is established during the execution of DH and ECDH key agreement algorithms:

- BCRYPT_KDF_SP80056A_CONCAT. This KDF supports the Concatenation KDF as specified in SP 800-56A (Section 5.8.1).
- BCRYPT_KDF_HASH. This KDF supports FIPS approved SP800-56A (Section 5.8), X9.63, and X9.42 key derivation.
- BCRYPT_KDF_HMAC. This KDF supports the IPsec IKEv1 key derivation that is non-Approved but is an allowed legacy implementation in FIPS mode when used to establish keys for IKEv1 as per scenario 4 of IG D.8.

Kernel Mode Cryptographic Primitives Library can use the following FIPS Approved key derivation functions (KDF) from a specified secret or password:

- BCRYPT_SP80056A_CONCAT_ALGORITHM. This KDF supports the Concatenation KDF as specified in SP 800-56Ar2 (Section 5.8.1).
- BCRYPT_SP800108_CTR_HMAC_ALGORITHM. This KDF supports the counter-mode variant of the KDF specified in SP 800-108 (Section 5.1) with HMAC as the underlying PRF.
- BCRYPT_PBKDF2_ALGORITHM. This KDF supports the Password Based Key Derivation Function specified in SP 800-132 (Section 5.3).

In addition the. the proprietary KDF, BCRYPT_CAPI_KDF_ALGORITHM is described at https://msdn.microsoft.com/library/windows/desktop/aa379916.aspx. This KDF cannot be used in a FIPS approved mode.

7.4.1 NIST SP 800-132 Password Based Key Derivation Function (PBKDF)

There are two options presented in NIST SP 800-132, pages 8 – 10, that are used to derive the Data Protection Key (DPK) from the Master Key. With the Kernel Mode Cryptographic Primitives Library, it is up to the caller to select the option to generate/protect the DPK. For example, DPAPI uses option 2a. Kernel Mode Cryptographic Primitives Library provides all the building blocks for the caller to select the desired option.

The Kernel Mode Cryptographic Primitives Library supports the following HMAC hash functions as parameters for PBKDF:

- SHA-1 HMAC
- SHA-256 HMAC
- SHA-384 HMAC
- SHA-512 HMAC

Keys derived from passwords, as described in SP 800-132, may only be used for storage applications. In order to run in a FIPS Approved manner, strong passwords must be used and they may only be used for storage applications. The password/passphrase length is enforced by the caller of the PBKDF interfaces when the password/passphrase is created and not by this cryptographic module.[14]

7.4.2 NIST SP 800-38F AES Key Wrapping

As outlined in FIPS 140-2 IG, D.2 and D.9, AES key wrapping serves as a form of key transport, which in turn is a form of key establishment. This implementation of AES key wrapping is in accordance with NIST SP 800-38F Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping.

[14] The probability of guessing a password is determined by its length and complexity, an organization should define a policy for these based based their threat model, suh as the example guidance in NIST SP800-63b, Appendix A.

7.5 Key Entry and Output

Keys can be both exported and imported out of and into Kernel Mode Cryptographic Primitives Library via BCryptExportKey(), BCryptImportKey(), and BCryptImportKeyPair() functions.

Symmetric key entry and output can also be done by exchanging keys using the recipient's asymmetric public key via BCryptSecretAgreement() and BCryptDeriveKey() functions.

Exporting the RSA private key by supplying a blob type of BCRYPT_PRIVATE_KEY_BLOB, BCRYPT_RSAFULLPRIVATE_BLOB, or BCRYPT_RSAPRIVATE_BLOB to BCryptExportKey() is not allowed in FIPS mode.

7.6 Key Storage

Kernel Mode Cryptographic Primitives Library does not provide persistent storage of keys.

7.7 Key Archival

Kernel Mode Cryptographic Primitives Library does not directly archive cryptographic keys. A user may choose to export a cryptographic key (cf. "Key Entry and Output" above), but management of the secure archival of that key is the responsibility of the user. All key copies inside Kernel Mode Cryptographic Primitives Library are destroyed and their memory location zeroized after used. It is the caller's responsibility to maintain the security of keys when the keys are outside Kernel Mode Cryptographic Primitives Library.

7.8 Key Zeroization

All keys are destroyed and their memory location zeroized when the operator calls BCryptDestroyKey() or BCryptDestroySecret() on that key handle.

8 Self-Tests

8.1 Power-On Self-Tests

The Kernel Mode Cryptographic Primitives Library module implements Known Answer Test (KAT) functions when the module is loaded into ntoskrnl.exe at boot time and the default driver entry point, DriverEntry, is called.

Kernel Mode Cryptographic Primitives Library performs the following power-on (startup) self-tests:

- HMAC (SHA-1, SHA-256, and SHA-512) Known Answer Tests
- Triple-DES encrypt/decrypt ECB Known Answer Tests
- AES-128 encrypt/decrypt ECB Known Answer Tests
- AES-128 encrypt/decrypt CCM Known Answer Tests
- AES-128 encrypt/decrypt CBC Known Answer Tests
- AES-128 CMAC Known Answer Test
- AES-128 encrypt/decrypt GCM Known Answer Tests
- XTS-AES encrypt/decrypt Known Answer Tests

- RSA sign/verify Known Answer Tests using RSA_SHA256_PKCS1 signature generation and verification
- ECDSA sign/verify Known Answer Tests on P256 curve
- DH secret agreement Known Answer Test with 2048-bit key
- ECDH secret agreement Known Answer Test on P256 curve
- SP 800-90A AES-256 counter mode DRBG Known Answer Tests (instantiate, generate and reseed)
- SP 800-108 KDF Known Answer Test
- SP 800-132 PBKDF Known Answer Test
- SHA-256 Known Answer Test
- SHA-512 Known Answer Test
- SP800-135 TLS 1.0/1.1 KDF Known Answer Test
- SP800-135 TLS 1.2 KDF Known Answer Test
- IKE SP800_135 KDF Known Answer Test

In any self-test fails, the Kernel Mode Cryptographic Primitives Library module does not load, an error code is returned to ntoskrnl.exe, and the computer will fail to boot.

8.2 Conditional Self-Tests

Kernel Mode Cryptographic Primitives Library performs pair-wise consistency checks upon each invocation of RSA, ECDH, and ECDSA key-pair generation and import as defined in FIPS 140-2.

ECDH key usage assurances are performed according to NIST SP 800-56A sections 5.5.2, 5.6.2, and 5.6.3.

A Continuous Random Number Generator Test (CRNGT) is performed for SP 800-90A AES-256 CTR DRBG per SP800-90A section 11.3.

A CRNGT is performed for the non-approved NDRNG per FIPS 140-2 IG 9.8.

When BCRYPT_ENABLE_INCOMPATIBLE_FIPS_CHECKS flag (required by policy) is used with BCryptGenerateSymmetricKey, then the XTS-AES Key_1 ≠ Key_2 check is performed in compliance with FIPS 140-2 IG A.9.

If the conditional self-test fails the function returns the status code STATUS_INTERNAL_ERROR.

9 Design Assurance

The secure installation, generation, and startup procedures of this cryptographic module are part of the overall operating system secure installation, configuration, and startup procedures for the Windows 10 operating system.

The Windows 10 operating system must be pre-installed on a computer by an OEM, installed by the end-user, by an organization's IT administrator, or updated from a previous Windows 10 version downloaded from Windows Update.

An inspection of authenticity of the physical medium can be made by following the guidance at this Microsoft web site: https://www.microsoft.com/en-us/howtotell/default.aspx

The installed version of Windows 10 must be verified to match the version that was validated. See Appendix A – How to Verify Windows Versions and Digital Signatures for details on how to do this.

For Windows Updates, the client only accepts binaries signed by Microsoft certificates. The Windows Update client only accepts content whose SHA-2 hash matches the SHA-2 hash specified in the metadata. All metadata communication is done over a Secure Sockets Layer (SSL) port. Using SSL ensures that the client is communicating with the real server and so prevents a spoof server from sending the client harmful requests. The version and digital signature of new cryptographic module releases must be verified to match the version that was validated. See Appendix A – How to Verify Windows Versions and Digital Signatures for details on how to do this.

10 Mitigation of Other Attacks

The following table lists the mitigations of other attacks for this cryptographic module:

Algorithm	Protected Against	Mitigation
SHA1	Timing Analysis Attack	Constant time implementation
	Cache Attack	Memory access pattern is independent of any confidential data
SHA2	Timing Analysis Attack	Constant time implementation
	Cache Attack	Memory access pattern is independent of any confidential data
Triple-DES	Timing Analysis Attack	Constant time implementation
AES	Timing Analysis Attack	Constant time implementation
	Cache Attack	Memory access pattern is independent of any confidential data
		Protected against cache attacks only when used with AES NI

11 Security Levels

The security level for each FIPS 140-2 security requirement is given in the following table.

Security Requirement	Security Level
Cryptographic Module Specification	1
Cryptographic Module Ports and Interfaces	1
Roles, Services, and Authentication	1
Finite State Model	1
Physical Security	NA
Operational Environment	1
Cryptographic Key Management	1
EMI/EMC	1
Self-Tests	1
Design Assurance	2
Mitigation of Other Attacks	1

12 Additional Details

For the latest information on Microsoft Windows, check out the Microsoft web site at:

https://www.microsoft.com/en-us/windows

For more information about FIPS 140 validations of Microsoft products, please see:

https://technet.microsoft.com/en-us/library/cc750357.aspx

13 Appendix A – How to Verify Windows Versions and Digital Signatures

13.1 How to Verify Windows Versions

The installed version of Windows 10 OEs must be verified to match the version that was validated using the following method:

1. In the Search box type "cmd" and open the Command Prompt desktop app.
2. The command window will open.
3. At the prompt, enter "ver”.
4. The version information will be displayed in a format like this:
   `Microsoft Windows [Version 10.0.xxxxx]`

If the version number reported by the utility matches the expected output, then the installed version has been validated to be correct.

13.2 How to Verify Windows Digital Signatures

After performing a Windows Update that includes changes to a cryptographic module, the digital signature and file version of the binary executable file must be verified. This is done like so:

1. Open a new window in Windows Explorer.
2. Type “C:\Windows\” in the file path field at the top of the window.
3. Type the cryptographic module binary executable file name (for example, “CNG.SYS”) in the search field at the top right of the window, then press the Enter key.
4. The file will appear in the window.
5. Right click on the file’s icon.
6. Select Properties from the menu and the Properties window opens.
7. Select the Details tab.
8. Note the File version Property and its value, which has a number in this format: xx.x.xxxxx.xxxx.
9. If the file version number matches one of the version numbers that appear at the start of this security policy document, then the version number has been verified.
10. Select the Digital Signatures tab.
11. In the Signature list, select the Microsoft Windows signer.
12. Click the Details button.
13. Under the Digital Signature Information, you should see: “This digital signature is OK.” If that condition is true, then the digital signature has been verified.

14 Appendix B – References

This table lists the specifications for each elliptic curve in section 2.3

Curve	Specification
Curve25519	https://cr.yp.to/ecdh/curve25519-20060209.pdf
brainpoolP160r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP192r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP192t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP224r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP224t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP256r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP256t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP320r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP320t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP384r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP384t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP512r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP512t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
ec192wapi	http://www.gbstandards.org/GB_standards/GB_standard.asp?id=900 (The GB standard is available here for purchase)
nistP192	http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf
nistP224	http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf
numsP256t1	https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/curvegen.pdf
numsP384t1	https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/curvegen.pdf
numsP512t1	https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/curvegen.pdf
secP160k1	http://www.secg.org/sec2-v2.pdf
secP160r1	http://www.secg.org/sec2-v2.pdf
secP160r2	http://www.secg.org/sec2-v2.pdf
secP192k1	http://www.secg.org/sec2-v2.pdf
secP192r1	http://www.secg.org/sec2-v2.pdf
secP224k1	http://www.secg.org/sec2-v2.pdf
secP224r1	http://www.secg.org/sec2-v2.pdf
secP256k1	http://www.secg.org/sec2-v2.pdf
secP256r1	http://www.secg.org/sec2-v2.pdf
secP384r1	http://www.secg.org/sec2-v2.pdf
secP521r1	http://www.secg.org/sec2-v2.pdf
wtls12	http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf
wtls7	http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf
wtls9	http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf

Curve	Specification
x962P192v1	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P192v2	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P192v3	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P239v1	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P239v2	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P239v3	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P256v1	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)

EXHIBIT J

Microsoft Windows 7 Cryptographic Primitives Library (bcryptprimitives.dll) Security Policy Document

Microsoft Windows 7 Operating System

FIPS 140-2 Security Policy Document

This document specifies the security policy for the Microsoft Windows Cryptographic Primitives Library (BCRYPTPRIMITIVES.DLL) as described in FIPS PUB 140-2.

May 2, 2011

Document Version: 2.2

1 Cryptographic Module Specification

The Microsoft Windows Cryptographic Primitives Library is a general purpose, software-based, cryptographic module. The primitive provider functionality is offered through one cryptographic module, BCRYPTPRIMITIVES.DLL (versions 6.1.7600.16385 and 6.1.7601.17514), subject to FIPS-140-2 validation. BCRYPTPRIMITIVES.DLL provides cryptographic services, through its documented interfaces, to Windows 7 components and applications running on Windows 7.

The cryptographic module, BCRYPTPRIMITIVES.DLL, encapsulates several different cryptographic algorithms in an easy-to-use cryptographic module accessible via the Microsoft CNG (Cryptography, Next Generation) API. It can be dynamically linked into applications by software developers to permit the use of general-purpose FIPS 140-2 Level 1 compliant cryptography.

1.1 Cryptographic Boundary

The Windows 7 BCRYPTPRIMITIVES.DLL consists of a dynamically-linked library (DLL). The cryptographic boundary for BCRYPTPRIMITIVES.DLL is defined as the enclosure of the computer system, on which BCRYPTPRIMITIVES.DLL is to be executed. The physical configuration of BCRYPTPRIMITIVES.DLL, as defined in FIPS-140-2, is multi-chip standalone.

2 Security Policy

BCRYPTPRIMITIVES.DLL operates under several rules that encapsulate its security policy.

- BCRYPTPRIMITIVES.DLL is supported on Windows 7 and Windows 7 SP1.
- BCRYPTPRIMITIVES.DLL operates in FIPS mode of operation only when used with the FIPS approved version of Windows 7 Code Integrity (ci.dll) validated to FIPS 140-2 under Cert. #1327 operating in FIPS mode
- Windows 7 is an operating system supporting a “single user” mode where there is only one interactive user during a logon session.
- BCRYPTPRIMITIVES.DLL is only in its Approved mode of operation when Windows is booted normally, meaning Debug mode is disabled and Driver Signing enforcement is enabled.
- BCRYPTPRIMITIVES.DLL operates in its FIPS mode of operation only when one of the following DWORD registry values is set to 1:
    - HKLM\SYSTEM\CurrentControlSet\Control\Lsa\FIPSAlgorithmPolicy\Enabled
    - HKLM\SYSTEM\CurrentControlSet\Policies\Microsoft\Cryptography\Configuration\SelfTest Algorithms
- All users assume either the User or Cryptographic Officer roles.
- BCRYPTPRIMITIVES.DLL provides no authentication of users. Roles are assumed implicitly. The authentication provided by the Windows 7 operating system is not in the scope of the validation.
- All cryptographic services implemented within BCRYPTPRIMITIVES.DLL are available to the User and Cryptographic Officer roles.
- BCRYPTPRIMITIVES.DLL implements the following FIPS-140-2 Approved algorithms:
    - SHA-1, SHA-256, SHA-384, SHA-512 hash (Cert. #1081)
    - SHA-1, SHA-256, SHA-384, SHA-512 HMAC (Cert. #677)
    - Triple-DES (2 key and 3 key) in ECB, CBC, and CFB8 modes (Cert. #846)
    - AES-128, AES-192, AES-256 in ECB, CBC, and CFB8 modes (Cert. #1168)
    - AES-128, AES-192 and AES-256 CCM (Cert. #1178)
    - AES-128, AES-192 and AES-256 GCM (Cert. #1168, vendor-affirmed)
    - AES-128, AES-192 and AES-256 GMAC (Cert#1168, vendor-affirmed)
    - RSA (RSASSA-PKCS1-v1_5 and RSASSA-PSS) digital signatures (Cert. #560) and X9.31 RSA key-pair generation (Cert. #559)
    - DSA (Cert. #386)
    - KAS – SP800-56A (vendor-affirmed) Diffie-Hellman Key Agreement; key establishment methodology provides at least 80-bits of encryption strength.

    - KAS – SP800-56A (vendor-affirmed) EC Diffie-Hellman Key Agreement; key establishment methodology provides between 128 and 256-bits of encryption strength
    - ECDSA with the following NIST curves: P-256, P-384, P-521 (Cert. #141)
    - SP800-90 AES-256 counter mode DRBG (Cert. #23)
    - SP800-90 Dual EC DRBG (Cert. #24)
    - FIPS 186-2 x-Change Notice Regular RNG (Cert. #649).
- BCRYPTPRIMITIVES.DLL supports the following non-Approved algorithms allowed for use in FIPS mode.
    - AES Key Wrap (AES Cert #1168, key wrapping; key establishment methodology provides 128 to 256 bits of encryption strength)
    - TLS and EAP-TLS
    - IKEv1 Key Derivation Functions
- BCRYPTPRIMITIVES.DLL also supports the following non FIPS 140-2 approved algorithms, though these algorithms may not be used when operating the module in a FIPS compliant manner.
    - RSA encrypt/decrypt
    - RC2, RC4, MD2, MD4, MD5, HMAC MD5[1].
    - DES in ECB, CBC, and CFB with 8-bit feedback

The following diagram illustrates the master components of the BCRYPTPRIMITIVES.DLL module

[1] Applications may not use any of these non-FIPS algorithms if they need to be FIPS compliant. To operate the module in a FIPS compliant manner, applications must only use FIPS-approved algorithms.

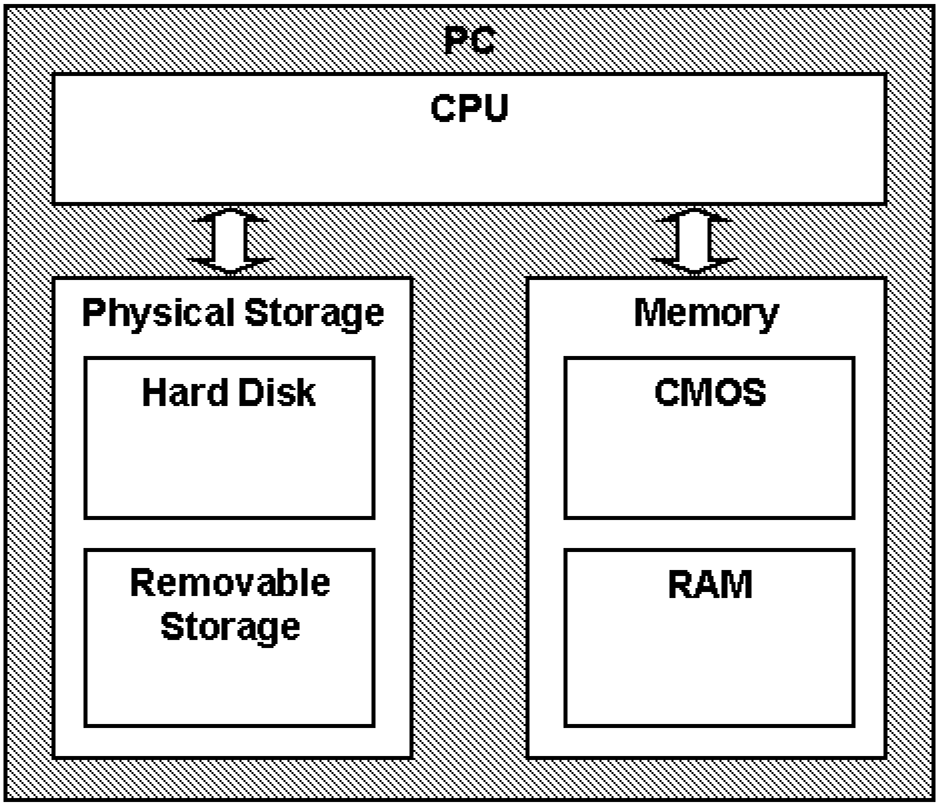


Figure 1 Master components of bcryptprimitives.dll module

BCRYPTPRIMITIVES.DLL (version: 6.1.7600.16385) was tested using the following machine configurations:

x86	Microsoft Windows 7 Ultimate Edition (x86 version) – HP Compaq dc7600
x64	Microsoft Windows 7 Ultimate Edition (x64 version) – HP Compaq dc7600

BCRYPTPRIMITIVES.DLL (version: 6.1.7601.17514) was tested using the following machine configurations:

x86	Microsoft Windows 7 Ultimate Edition (x86 version) – HP Compaq dc7600
x64	Microsoft Windows 7 Ultimate Edition (x64 version) – HP Compaq dc7600

3 Cryptographic Module Ports and Interfaces

3.1 Ports and Interfaces

3.1.1 Export Functions

The BCRYPTPRIMITIVES.DLL module implements a set of algorithm providers for the Cryptography Next Generation (CNG) framework in Windows. Each provider in this module represents a single cryptographic algorithm or a set of closely related cryptographic algorithms. These algorithm providers are invoked through the CNG algorithm primitive functions, which are sometimes collectively referred to as the BCrypt API. For a full list of these algorithm providers, see http://msdn.microsoft.com/en-us/library/aa375534.aspx.

The BCRYPTPRIMITIVES.DLL module exposes its cryptographic services to the operating system through a small set of exported functions. These functions are used by the CNG framework to retrieve references to the different algorithm providers, in order to route BCrypt API calls appropriately to BCRYPTPRIMITIVES.DLL. For details, please see the CNG SDK, available at http://www.microsoft.com/downloads/details.aspx?familyid=1EF399E9-B018-49DB-A98B-0CED7CB8FF6F&displaylang=en.

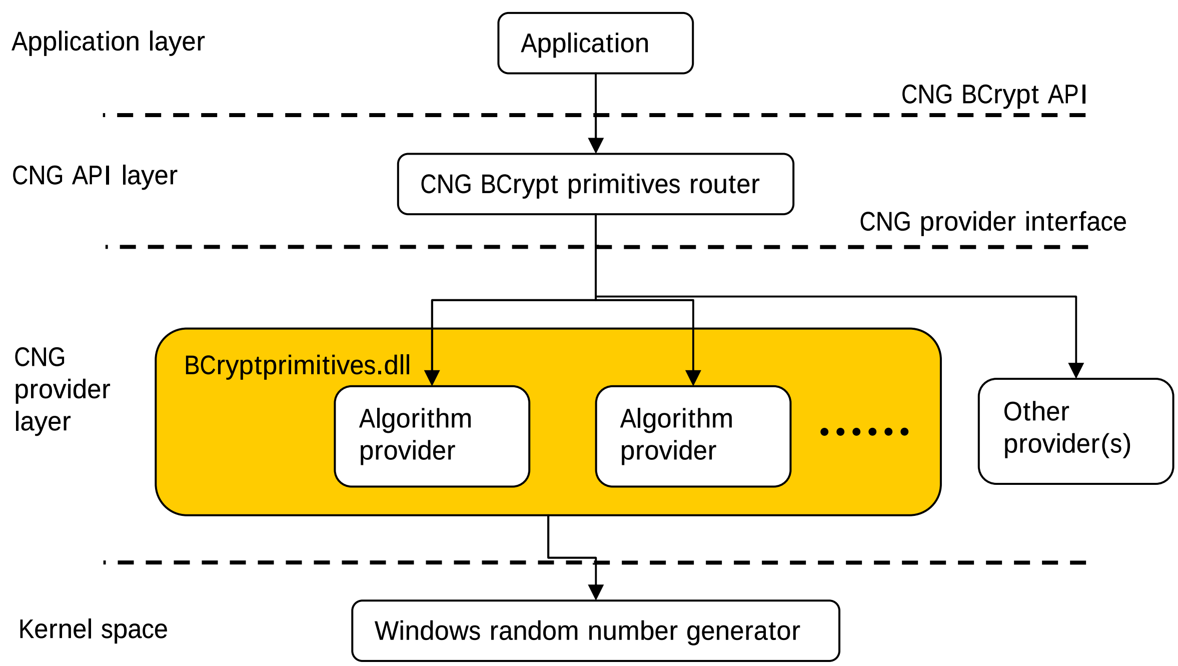


Figure 2 Relationships between bcryptprimitives.dll and other components – cryptographic boundary highlighted in gold.

The following functions are exported by BCRYPTPRIMITIVES.DLL:

- GetAsymmetricEncryptionInterface
- GetCipherInterface
- GetHashInterface
- GetRngInterface
- GetSecretAgreementInterface
- GetSignatureInterface

3.1.2 CNG Primitive Functions

The following list contains the CNG functions which can be used by callers to access the cryptographic services in BCRYPTPRIMITIVES.DLL.

- BCryptCloseAlgorithmProvider
- BCryptCreateHash
- BCryptDecrypt
- BCryptDeriveKey
- BCryptDestroyHash
- BCryptDestroyKey

- BCryptDestroySecret
- BCryptDuplicateHash
- BCryptDuplicateKey
- BCryptEncrypt
- BCryptExportKey
- BCryptFinalizeKeyPair
- BCryptFinishHash
- BCryptFreeBuffer
- BCryptGenerateKeyPair
- BCryptGenerateSymmetricKey
- BCryptGenRandom
- BCryptGetProperty
- BCryptHashData
- BCryptImportKey
- BCryptImportKeyPair
- BCryptOpenAlgorithmProvider
- BCryptSecretAgreement
- BCryptSetProperty
- BCryptSignHash
- BCryptVerifySignature

3.1.3 Data Input and Output Interfaces

The Data Input Interface for BCRYPTPRIMITIVES.DLL consists of the CNG primitive functions listed in Section 3.1.2. Data and options are passed to the interface as input parameters to the CNG primitive functions. Data Input is kept separate from Control Input by passing Data Input in separate parameters from Control Input.

The Data Output Interface for BCRYPTPRIMITIVES.DLL also consists of the CNG primitive functions.

3.1.4 Control Input Interface

The Control Input Interface for BCRYPTPRIMITIVES.DLL also consists of the CNG primitive functions. Options for control operations are passed as input parameters to the CNG primitive functions.

3.1.5 Status Output Interface

The Status Output Interface for BCRYPTPRIMITIVES.DLL also consists of the CNG primitive functions. For each function, the status information is returned to the caller as the return value from the function.

3.2 Cryptographic Bypass

Cryptographic bypass is not supported by BCRYPTPRIMITIVES.DLL.

4 Roles and Authentication

4.1 Roles

BCRYPTPRIMITIVES.DLL provides User and Cryptographic Officer roles (as defined in FIPS 140-2). These roles share all the services implemented in the cryptographic module.
When an application requests the crypto module to generate keys for a user, the keys are generated, used, and deleted as requested by applications. There are no implicit keys associated with a user. Each user may have numerous keys, and each user's keys are separate from other users' keys.

4.2 Maintenance Roles

Maintenance roles are not supported by BCRYPTPRIMITIVES.DLL.

4.3 Operator Authentication

The module does not provide authentication. Roles are implicitly assumed based on the services that are executed.

The OS on which BCRYPTPRIMITIVES.DLL executes (Microsoft Windows 7) does authenticate users. Microsoft Windows 7 requires authentication from the trusted control base (TCB) before a user is able to access system services. Once a user is authenticated from the TCB, a process is created bearing the Authenticated User's security token for identification purpose. All subsequent processes and threads created by that Authenticated User are implicitly assigned the parent's (thus the Authenticated User's) security token.

5 Services

The following list contains all services available to an operator. All services are accessible to both the User and Crypto Officer roles.

5.1 Algorithm Providers and Properties

5.1.1 BCryptOpenAlgorithmProvider

```
NTSTATUS WINAPI BCryptOpenAlgorithmProvider(
        BCRYPT_ALG_HANDLE   *phAlgorithm,
        LPCWSTR pszAlgId,
        LPCWSTR pszImplementation,
        ULONG   dwFlags);
```

The BCryptOpenAlgorithmProvider() function has four parameters: algorithm handle output to the opened algorithm provider, desired algorithm ID input, an optional specific provider name input, and optional flags. This function loads and initializes a CNG provider for a given algorithm, and returns a handle to the opened algorithm provider on success. See http://msdn.microsoft.com for CNG providers. Unless the calling function specifies the name of the provider, the default provider is used. The default provider is the first provider listed for a given algorithm. The calling function must pass the BCRYPT_ALG_HANDLE_HMAC_FLAG flag in order to use an HMAC function with a hash algorithm.

5.1.2 BCryptCloseAlgorithmProvider

```
NTSTATUS WINAPI BCryptCloseAlgorithmProvider(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        ULONG   dwFlags);
```

This function closes an algorithm provider handle opened by a call to BCryptOpenAlgorithmProvider() function.

5.1.3 BCryptSetProperty

```
NTSTATUS WINAPI BCryptSetProperty(
        BCRYPT_HANDLE   hObject,
        LPCWSTR pszProperty,
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptSetProperty() function sets the value of a named property for a CNG object, e.g., a cryptographic key. The CNG object is referenced by a handle, the property name is a NULL terminated string, and the value of the property is a length-specified byte string.

5.1.4 BCryptGetProperty

```
NTSTATUS WINAPI BCryptGetProperty(
        BCRYPT_HANDLE   hObject,
        LPCWSTR pszProperty,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptGetProperty() function retrieves the value of a named property for a CNG object, e.g., a cryptographic key. The CNG object is referenced by a handle, the property name is a NULL terminated string, and the value of the property is a length-specified byte string.

5.1.5 BCryptFreeBuffer

```
VOID WINAPI BCryptFreeBuffer(
        PVOID   pvBuffer);
```

Some of the CNG functions allocate memory on caller's behalf. The BCryptFreeBuffer() function frees memory that was allocated by such a CNG function.

5.2 Random Number Generation

5.2.1 BCryptGenRandom

```
NTSTATUS WINAPI BCryptGenRandom(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        PUCHAR  pbBuffer,
        ULONG   cbBuffer,
        ULONG   dwFlags);
```

The BCryptGenRandom() function fills a buffer with random bytes. BCRYPTPRIMITVES.DLL implements three random number generation algorithms:

- BCRYPT_RNG_ALGORITHM. This is the AES-256 counter mode based random generator as defined in SP800-90.
- BCRYPT_RNG_DUAL_EC_ALGORITHM. This is the dual elliptic curve based random generator as defined in SP800-90.
- BCRYPT_RNG_FIPS186_DSA_ALGORITHM. This is the random number generator required by the DSA algorithm as defined in FIPS 186-2.

When BCRYPT_RNG_USE_ENTROPY_IN_BUFFER is specified in the *dwFlags* parameter, this function will use the number in the *pbBuffer* buffer as additional entropy for the random number generation algorithm.

During the function initialization, a seed is obtained from the output of an in-kernel random number generator. This RNG, which exists beyond the cryptographic boundary, provides the necessary entropy for the user-level RNGs available through this function.

5.3 Key and Key-Pair Generation

5.3.1 BCryptGenerateSymmetricKey

```
NTSTATUS WINAPI BCryptGenerateSymmetricKey(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        BCRYPT_KEY_HANDLE   *phKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        PUCHAR   pbSecret,
        ULONG   cbSecret,
        ULONG   dwFlags);
```

The BCryptGenerateSymmetricKey() function generates a symmetric key object for use with a symmetric encryption algorithm from a supplied *cbSecret* bytes long key value provided in the *pbSecret* memory location. The calling application must specify a handle to the algorithm provider opened with the BCryptOpenAlgorithmProvider() function. The algorithm specified when the provider was opened must support symmetric key encryption.

5.3.2 BCryptGenerateKeyPair

```
NTSTATUS WINAPI BCryptGenerateKeyPair(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        BCRYPT_KEY_HANDLE   *phKey,
        ULONG   dwLength,
        ULONG   dwFlags);
```

The BCryptGenerateKeyPair() function creates a public/private key pair object without any cryptographic keys in it. After creating such an empty key pair object using this function, call the BCryptSetProperty() function to set its properties. The key pair can be used only after BCryptFinalizeKeyPair() function is called.

5.3.3 BCryptFinalizeKeyPair

```
NTSTATUS WINAPI BCryptFinalizeKeyPair(
        BCRYPT_KEY_HANDLE   hKey,
        ULONG   dwFlags);
```

The BCryptFinalizeKeyPair() function completes a public/private key pair import or generation. The key pair cannot be used until this function has been called. After this function has been called, the BCryptSetProperty() function can no longer be used for this key pair.

5.3.4 BCryptDuplicateKey

```
NTSTATUS WINAPI BCryptDuplicateKey(
        BCRYPT_KEY_HANDLE   hKey,
        BCRYPT_KEY_HANDLE   *phNewKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        ULONG   dwFlags);
```

The BCryptDuplicateKey() function creates a duplicate of a symmetric key object.

5.3.5 BCryptDestroyKey

```
NTSTATUS WINAPI BCryptDestroyKey(
        BCRYPT_KEY_HANDLE   hKey);
```

The BCryptDestroyKey() function destroys a key.

5.4 Key Entry and Output

5.4.1 BCryptImportKey

```
NTSTATUS WINAPI BCryptImportKey(
        BCRYPT_ALG_HANDLE hAlgorithm,
        BCRYPT_KEY_HANDLE hImportKey,
        LPCWSTR pszBlobType,
        BCRYPT_KEY_HANDLE *phKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptImportKey() function imports a symmetric key from a key blob.

hAlgorithm [in] is the handle of the algorithm provider to import the key. This handle is obtained by calling the BCryptOpenAlgorithmProvider function.
hImportKey [in, out] is not currently used and should be NULL.
pszBlobType [in] is a null-terminated Unicode string that contains an identifier that specifies the type of BLOB that is contained in the *pbInput* buffer. *pszBlobType* can be one of BCRYPT_AES_WRAP_KEY_BLOB, BCRYPT_KEY_DATA_BLOB and BCRYPT_OPAQUE_KEY_BLOB.
phKey [out] is a pointer to a BCRYPT_KEY_HANDLE that receives the handle of the imported key that is used in subsequent functions that require a key, such as BCryptEncrypt. This handle must be released when it is no longer needed by passing it to the BCryptDestroyKey function.
pbKeyObject [out] is a pointer to a buffer that receives the imported key object. The *cbKeyObject* parameter contains the size of this buffer. The required size of this buffer can be obtained by calling the BCryptGetProperty function to get the BCRYPT_OBJECT_LENGTH property. This will provide the size of the key object for the specified algorithm. This memory can only be freed after the *phKey* key handle is destroyed.
cbKeyObject [in] is the size, in bytes, of the pbKeyObject buffer.
pbInput [in] is the address of a buffer that contains the key BLOB to import.
The *cbInput* parameter contains the size of this buffer.
The *pszBlobType* parameter specifies the type of key BLOB this buffer contains.
cbInput [in] is the size, in bytes, of the pbInput buffer.
dwFlags [in] is a set of flags that modify the behavior of this function. No flags are currently defined, so this parameter should be zero.

5.4.2 BCryptImportKeyPair

```
NTSTATUS WINAPI BCryptImportKeyPair(
        BCRYPT_ALG_HANDLE hAlgorithm,
        BCRYPT_KEY_HANDLE hImportKey,
        LPCWSTR pszBlobType,
        BCRYPT_KEY_HANDLE *phKey,
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptImportKeyPair() function is used to import a public/private key pair from a key blob.
hAlgorithm [in] is the handle of the algorithm provider to import the key. This handle is obtained by calling the BCryptOpenAlgorithmProvider function.
hImportKey [in, out] is not currently used and should be NULL.
pszBlobType [in] is a null-terminated Unicode string that contains an identifier that specifies the type of BLOB that is contained in the pbInput buffer. This can be one of the following values: BCRYPT_DH_PRIVATE_BLOB, BCRYPT_DH_PUBLIC_BLOB, BCRYPT_DSA_PRIVATE_BLOB, BCRYPT_DSA_PUBLIC_BLOB, BCRYPT_ECCPRIVATE_BLOB, BCRYPT_ECCPUBLIC_BLOB, BCRYPT_PUBLIC_KEY_BLOB, BCRYPT_PRIVATE_KEY_BLOB, BCRYPT_RSAPRIVATE_BLOB, BCRYPT_RSAPUBLIC_BLOB, LEGACY_DH_PUBLIC_BLOB, LEGACY_DH_PRIVATE_BLOB, LEGACY_DSA_PRIVATE_BLOB, LEGACY_DSA_PUBLIC_BLOB, LEGACY_DSA_V2_PRIVATE_BLOB, LEGACY_RSAPRIVATE_BLOB, LEGACY_RSAPUBLIC_BLOB.
phKey [out] is a pointer to a BCRYPT_KEY_HANDLE that receives the handle of the imported key. This handle is used in subsequent functions that require a key, such as BCryptSignHash. This handle must be released when it is no longer needed by passing it to the BCryptDestroyKey function.
pbInput [in] is the address of a buffer that contains the key BLOB to import. The cbInput parameter contains the size of this buffer. The pszBlobType parameter specifies the type of key BLOB this buffer contains.

cbInput [in] contains the size, in bytes, of the pbInput buffer.
dwFlags [in] is a set of flags that modify the behavior of this function. This can be zero or the following value: BCRYPT_NO_KEY_VALIDATION.

5.4.3 BCryptExportKey

```
NTSTATUS WINAPI BCryptExportKey(
        BCRYPT_KEY_HANDLE   hKey,
        BCRYPT_KEY_HANDLE   hExportKey,
        LPCWSTR pszBlobType,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptExportKey() function exports a key to a memory blob that can be persisted for later use.
hKey [in] is the handle of the key to export.
hExportKey [in, out] is not currently used and should be set to NULL.
pszBlobType [in] is a null-terminated Unicode string that contains an identifier that specifies the type of BLOB to export. This can be one of the following values: BCRYPT_AES_WRAP_KEY_BLOB, BCRYPT_DH_PRIVATE_BLOB, BCRYPT_DH_PUBLIC_BLOB, BCRYPT_DSA_PRIVATE_BLOB, BCRYPT_DSA_PUBLIC_BLOB, BCRYPT_ECCPRIVATE_BLOB, BCRYPT_ECCPUBLIC_BLOB, BCRYPT_KEY_DATA_BLOB, BCRYPT_OPAQUE_KEY_BLOB, BCRYPT_PUBLIC_KEY_BLOB, BCRYPT_PRIVATE_KEY_BLOB, BCRYPT_RSAPRIVATE_BLOB, BCRYPT_RSAPUBLIC_BLOB, LEGACY_DH_PRIVATE_BLOB, LEGACY_DH_PUBLIC_BLOB, LEGACY_DSA_PRIVATE_BLOB, LEGACY_DSA_PUBLIC_BLOB, LEGACY_DSA_V2_PRIVATE_BLOB, LEGACY_RSAPRIVATE_BLOB, LEGACY_RSAPUBLIC_BLOB.
pbOutput is the address of a buffer that receives the key BLOB. The cbOutput parameter contains the size of this buffer. If this parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by the pcbResult parameter.
cbOutput [in] contains the size, in bytes, of the pbOutput buffer.
pcbResult [out] is a pointer to a ULONG that receives the number of bytes that were copied to the pbOutput buffer. If the pbOutput parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by this parameter.
dwFlags [in] is a set of flags that modify the behavior of this function. No flags are defined for this function.

5.5 Encryption and Decryption

5.5.1 BCryptEncrypt

```
NTSTATUS WINAPI BCryptEncrypt(
        BCRYPT_KEY_HANDLE hKey,
        PUCHAR   pbInput,
        ULONG   cbInput,
        VOID    *pPaddingInfo,
        PUCHAR   pbIV,
        ULONG   cbIV,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptEncrypt() function encrypts a block of data of given length.
hKey [in, out] is the handle of the key to use to encrypt the data. This handle is obtained from one of the key creation functions, such as BCryptGenerateSymmetricKey, BCryptGenerateKeyPair, or BCryptImportKey.

pbInput [in] is the address of a buffer that contains the plaintext to be encrypted. The cbInput parameter contains the size of the plaintext to encrypt. For more information, see Remarks.
cbInput [in] is the number of bytes in the pbInput buffer to encrypt.
pPaddingInfo [in, optional] is a pointer to a structure that contains padding information. The actual type of structure this parameter points to depends on the value of the dwFlags parameter. This parameter is only used with asymmetric keys and authenticated encryption modes (i.e. AES-CCM and AES-GCM). It must be NULL otherwise.
pbIV [in, out, optional] is the address of a buffer that contains the initialization vector (IV) to use during encryption. The cbIV parameter contains the size of this buffer. This function will modify the contents of this buffer. If you need to reuse the IV later, make sure you make a copy of this buffer before calling this function. This parameter is optional and can be NULL if no IV is used. The required size of the IV can be obtained by calling the BCryptGetProperty function to get the BCRYPT_BLOCK_LENGTH property. This will provide the size of a block for the algorithm, which is also the size of the IV.
cbIV [in] contains the size, in bytes, of the pbIV buffer.
pbOutput [out, optional] is the address of a buffer that will receive the ciphertext produced by this function. The cbOutput parameter contains the size of this buffer. For more information, see Remarks. If this parameter is NULL, this function will calculate the size needed for the ciphertext and return the size in the location pointed to by the pcbResult parameter.
cbOutput [in] contains the size, in bytes, of the pbOutput buffer. This parameter is ignored if the pbOutput parameter is NULL.
pcbResult [out] is a pointer to a ULONG variable that receives the number of bytes copied to the pbOutput buffer. If pbOutput is NULL, this receives the size, in bytes, required for the ciphertext.
dwFlags [in] is a set of flags that modify the behavior of this function. The allowed set of flags depends on the type of key specified by the hKey parameter. If the key is a symmetric key, this can be zero or the following value: BCRYPT_BLOCK_PADDING. If the key is an asymmetric key, this can be one of the following values: BCRYPT_PAD_NONE, BCRYPT_PAD_OAEP, BCRYPT_PAD_PKCS1.

5.5.2 BCryptDecrypt

```
NTSTATUS WINAPI BCryptDecrypt(
        BCRYPT_KEY_HANDLE   hKey,
        PUCHAR   pbInput,
        ULONG   cbInput,
        VOID    *pPaddingInfo,
        PUCHAR   pbIV,
        ULONG   cbIV,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptDecrypt() function decrypts a block of data of given length.
hKey [in, out] is the handle of the key to use to decrypt the data. This handle is obtained from one of the key creation functions, such as BCryptGenerateSymmetricKey, BCryptGenerateKeyPair, or BCryptImportKey.
pbInput [in] is the address of a buffer that contains the ciphertext to be decrypted. The cbInput parameter contains the size of the ciphertext to decrypt. For more information, see Remarks.
cbInput [in] is the number of bytes in the pbInput buffer to decrypt.
pPaddingInfo [in, optional] is a pointer to a structure that contains padding information. The actual type of structure this parameter points to depends on the value of the dwFlags parameter. This parameter is only used with asymmetric keys and authenticated encryption modes (i.e. AES-CCM and AES-GCM). It must be NULL otherwise.
pbIV [in, out, optional] is the address of a buffer that contains the initialization vector (IV) to use during decryption. The cbIV parameter contains the size of this buffer. This function will modify the contents of

this buffer. If you need to reuse the IV later, make sure you make a copy of this buffer before calling this function. This parameter is optional and can be NULL if no IV is used. The required size of the IV can be obtained by calling the BCryptGetProperty function to get the BCRYPT_BLOCK_LENGTH property. This will provide the size of a block for the algorithm, which is also the size of the IV.
cbIV [in] contains the size, in bytes, of the pbIV buffer.
pbOutput [out, optional] is the address of a buffer to receive the plaintext produced by this function. The cbOutput parameter contains the size of this buffer. For more information, see Remarks.
If this parameter is NULL, this function will calculate the size required for the plaintext and return the size in the location pointed to by the pcbResult parameter.
cbOutput [in] is the size, in bytes, of the pbOutput buffer. This parameter is ignored if the pbOutput parameter is NULL.
pcbResult [out] is a pointer to a ULONG variable to receive the number of bytes copied to the pbOutput buffer. If pbOutput is NULL, this receives the size, in bytes, required for the plaintext.
dwFlags [in] is a set of flags that modify the behavior of this function. The allowed set of flags depends on the type of key specified by the hKey parameter. If the key is a symmetric key, this can be zero or the following value: BCRYPT_BLOCK_PADDING. If the key is an asymmetric key, this can be one of the following values: BCRYPT_PAD_NONE, BCRYPT_PAD_OAEP, BCRYPT_PAD_PKCS1.

5.6 Hashing and Message Authentication

5.6.1 BCryptCreateHash

```
NTSTATUS WINAPI BCryptCreateHash(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        BCRYPT_HASH_HANDLE  *phHash,
        PUCHAR   pbHashObject,
        ULONG   cbHashObject,
        PUCHAR   pbSecret,
        ULONG   cbSecret,
        ULONG   dwFlags);
```

The BCryptCreateHash() function creates a hash object with an optional key. The optional key is used for HMAC and AES GMAC.
hAlgorithm [in, out] is the handle of an algorithm provider created by using the BCryptOpenAlgorithmProvider function. The algorithm that was specified when the provider was created must support the hash interface.
phHash [out] is a pointer to a BCRYPT_HASH_HANDLE value that receives a handle that represents the hash object. This handle is used in subsequent hashing functions, such as the BCryptHashData function. When you have finished using this handle, release it by passing it to the BCryptDestroyHash function.
pbHashObject [out] is a pointer to a buffer that receives the hash object. The cbHashObject parameter contains the size of this buffer. The required size of this buffer can be obtained by calling the BCryptGetProperty function to get the BCRYPT_OBJECT_LENGTH property. This will provide the size of the hash object for the specified algorithm. This memory can only be freed after the hash handle is destroyed.
cbHashObject [in] contains the size, in bytes, of the pbHashObject buffer.
pbSecret [in, optional] is a pointer to a buffer that contains the key to use for the hash. The cbSecret parameter contains the size of this buffer. If no key should be used with the hash, set this parameter to NULL. This key only applies to the HMAC and AES GMAC algorithms.
cbSecret [in, optional] contains the size, in bytes, of the pbSecret buffer. If no key should be used with the hash, set this parameter to zero.
dwFlags [in] is not currently used and must be zero.

5.6.2 BCryptHashData

```
NTSTATUS WINAPI BCryptHashData(
        BCRYPT_HASH_HANDLE  hHash,
```

```
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptHashData() function performs a one way hash on a data buffer. Call the BCryptFinishHash() function to finalize the hashing operation to get the hash result.

5.6.3 BCryptDuplicateHash

```
NTSTATUS WINAPI BCryptDuplicateHash(
        BCRYPT_HASH_HANDLE  hHash,
        BCRYPT_HASH_HANDLE  *phNewHash,
        PUCHAR   pbHashObject,
        ULONG   cbHashObject,
        ULONG   dwFlags);
```

The BCryptDuplicateHash()function duplicates an existing hash object. The duplicate hash object contains all state and data that was hashed to the point of duplication.

5.6.4 BCryptFinishHash

```
NTSTATUS WINAPI BCryptFinishHash(
        BCRYPT_HASH_HANDLE hHash,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   dwFlags);
```

The BCryptFinishHash() function retrieves the hash value for the data accumulated from prior calls to BCryptHashData() function.

5.6.5 BCryptDestroyHash

```
NTSTATUS WINAPI BCryptDestroyHash(
        BCRYPT_HASH_HANDLE  hHash);
```

The BCryptDestroyHash() function destroys a hash object.

5.7 Signing and Verification

5.7.1 BCryptSignHash

```
NTSTATUS WINAPI BCryptSignHash(
        BCRYPT_KEY_HANDLE   hKey,
        VOID    *pPaddingInfo,
        PUCHAR   pbInput,
        ULONG   cbInput,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptSignHash() function creates a signature of a hash value.
hKey [in] is the handle of the key to use to sign the hash.
pPaddingInfo [in, optional] is a pointer to a structure that contains padding information. The actual type of structure this parameter points to depends on the value of the dwFlags parameter. This parameter is only used with asymmetric keys and must be NULL otherwise.
pbInput [in] is a pointer to a buffer that contains the hash value to sign. The cbInput parameter contains the size of this buffer.
cbInput [in] is the number of bytes in the pbInput buffer to sign.
pbOutput [out] is the address of a buffer to receive the signature produced by this function. The cbOutput parameter contains the size of this buffer. If this parameter is NULL, this function will calculate

the size required for the signature and return the size in the location pointed to by the pcbResult parameter.
cbOutput [in] is the size, in bytes, of the pbOutput buffer. This parameter is ignored if the pbOutput parameter is NULL.
pcbResult [out] is a pointer to a ULONG variable that receives the number of bytes copied to the pbOutput buffer. If pbOutput is NULL, this receives the size, in bytes, required for the signature.
dwFlags [in] is a set of flags that modify the behavior of this function. The allowed set of flags depends on the type of key specified by the hKey parameter. If the key is a symmetric key, this parameter is not used and should be set to zero. If the key is an asymmetric key, this can be one of the following values: BCRYPT_PAD_PKCS1, BCRYPT_PAD_PSS.

5.7.2 BCryptVerifySignature

```
NTSTATUS WINAPI BCryptVerifySignature(
        BCRYPT_KEY_HANDLE   hKey,
        VOID    *pPaddingInfo,
        PUCHAR   pbHash,
        ULONG   cbHash,
        PUCHAR   pbSignature,
        ULONG   cbSignature,
        ULONG   dwFlags);
```

The BCryptVerifySignature() function verifies that the specified signature matches the specified hash.
hKey [in] is the handle of the key to use to decrypt the signature. This must be an identical key or the public key portion of the key pair used to sign the data with the BCryptSignHash function.
pPaddingInfo [in, optional] is a pointer to a structure that contains padding information. The actual type of structure this parameter points to depends on the value of the *dwFlags* parameter. This parameter is only used with asymmetric keys and must be NULL otherwise.
pbHash [in] is the address of a buffer that contains the hash of the data. The cbHash parameter contains the size of this buffer.
cbHash [in] is the size, in bytes, of the pbHash buffer.
pbSignature [in] is the address of a buffer that contains the signed hash of the data. The BCryptSignHash function is used to create the signature. The *cbSignature* parameter contains the size of this buffer.
cbSignature [in] is the size, in bytes, of the pbSignature buffer. The BCryptSignHash function is used to create the signature.

5.8 Secret Agreement and Key Derivation

5.8.1 BCryptSecretAgreement

```
NTSTATUS WINAPI BCryptSecretAgreement(
        BCRYPT_KEY_HANDLE       hPrivKey,
        BCRYPT_KEY_HANDLE       hPubKey,
        BCRYPT_SECRET_HANDLE    *phAgreedSecret,
        ULONG                   dwFlags);
```

The BCryptSecretAgreement() function creates a secret agreement value from a private and a public key. This function is used with Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH) algorithms.
hPrivKey [in] The handle of the private key to use to create the secret agreement value.
hPubKey [in] The handle of the public key to use to create the secret agreement value.
phSecret [out] A pointer to a BCRYPT_SECRET_HANDLE that receives a handle that represents the secret agreement value. This handle must be released by passing it to the BCryptDestroySecret function when it is no longer needed.
dwFlags [in] A set of flags that modify the behavior of this function. This must be zero.

5.8.2 BCryptDeriveKey

```
NTSTATUS WINAPI BCryptDeriveKey(
```

```
BCRYPT_SECRET_HANDLE hSharedSecret,
LPCWSTR              pwszKDF,
BCryptBufferDesc     *pParameterList,
PUCHAR pbDerivedKey,
ULONG                cbDerivedKey,
ULONG                *pcbResult,
ULONG                dwFlags);
```

The BCryptDeriveKey() function derives a key from a secret agreement value.
hSharedSecret [in, optional] is the secret agreement handle to create the key from. This handle is obtained from the BCryptSecretAgreement function.
pwszKDF [in] is a pointer to a null-terminated Unicode string that contains an object identifier (OID) that identifies the key derivation function (KDF) to use to derive the key. This can be one of the following strings: BCRYPT_KDF_HASH (parameters in pParameterList: KDF_HASH_ALGORITHM, KDF_SECRET_PREPEND, KDF_SECRET_APPEND), BCRYPT_KDF_HMAC (parameters in pParameterList: KDF_HASH_ALGORITHM, KDF_HMAC_KEY, KDF_SECRET_PREPEND, KDF_SECRET_APPEND), BCRYPT_KDF_TLS_PRF (parameters in pParameterList: KDF_TLS_PRF_LABEL, KDF_TLS_PRF_SEED), BCRYPT_KDF_SP80056A_CONCAT (parameters in pParameterList: KDF_ALGORITHMID, KDF_PARTYUINFO, KDF_PARTYVINFO, KDF_SUPPPUBINFO, KDF_SUPPPRIVINFO).
pParameterList [in, optional] is the address of a BCryptBufferDesc structure that contains the KDF parameters. This parameter is optional and can be NULL if it is not needed.
pbDerivedKey [out, optional] is the address of a buffer that receives the key. The cbDerivedKey parameter contains the size of this buffer. If this parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by the pcbResult parameter.
cbDerivedKey [in] contains the size, in bytes, of the pbDerivedKey buffer.
pcbResult [out] is a pointer to a ULONG that receives the number of bytes that were copied to the pbDerivedKey buffer. If the pbDerivedKey parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by this parameter.
dwFlags [in] is a set of flags that modify the behavior of this function. This can be zero or KDF_USE_SECRET_AS_HMAC_KEY_FLAG. The KDF_USE_SECRET_AS_HMAC_KEY_FLAG value must only be used when pwszKDF is equal to BCRYPT_KDF_HMAC. It indicates that the secret will also be used as the HMAC key. If this flag is used, the KDF_HMAC_KEY parameter must not be specified in pParameterList.

5.8.3 BCryptDestroySecret

```
NTSTATUS WINAPI BCryptDestroySecret(
    BCRYPT_SECRET_HANDLE   hSecret);
```

The BCryptDestroySecret() function destroys a secret agreement handle that was created by using the BCryptSecretAgreement() function.

6 Operational Environment

BCRYPTPRIMITIVES.DLL is intended to run on Windows 7 in Single User mode as defined in Section 2. When run in this configuration, multiple concurrent operators are not supported.
Because BCRYPTPRIMITIVES.DLL module is a DLL, each process requesting access is provided its own instance of the module. As such, each process has full access to all information and keys within the module. Note that no keys or other information are maintained upon detachment from the DLL, thus an instantiation of the module will only contain keys or information that the process has placed in the module.

7 Cryptographic Key Management

BCRYPTPRIMITIVES.DLL crypto module manages keys in the following manner.

7.1 Cryptographic Keys, CSPs, and SRDIs

The BCRYPTPRIMITIVES.DLL crypto module contains the following security relevant data items:

Security Relevant Data Item	SRDI Description
Symmetric encryption/decryption keys	Keys used for AES or TDEA encryption/decryption.
HMAC keys	Keys used for HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, and HMAC-SHA512
DSA Public Keys	Keys used for the verification of DSA digital signatures
DSA Private Keys	Keys used for the calculation of DSA digital signatures
ECDSA Public Keys	Keys used for the verification of ECDSA digital signatures
ECDSA Private Keys	Keys used for the calculation of ECDSA digital signatures
RSA Public Keys	Keys used for the verification of RSA digital signatures
RSA Private Keys	Keys used for the calculation of RSA digital signatures
DH Public and Private values	Public and private values used for Diffie-Hellman key establishment.
ECDH Public and Private values	Public and private values used for EC Diffie-Hellman key establishment.

7.2 Access Control Policy

The BCRYPTPRIMITIVES.DLL crypto module allows controlled access to the SRDIs contained within it. The following table defines the access that a service has to each. The permissions are categorized as a set of four separate permissions: read (r), write (w), execute (x), delete (d). If no permission is listed, the service has no access to the SRDI.

BCRYPTPRIMITIVES.DLL cryptographic module SRDI/Service Access Policy	Symmetric encryption and decryption keys	HMAC keys	DSA Public Keys	DSA Private Keys	ECDSA public keys	ECDSA Private keys	RSA Public Keys	RSA Private Keys	DH Public and Private values	ECDH Public and Private values
Cryptographic Module Power Up and Power Down										
Key Formatting	w									
Random Number Generation										
Data Encryption and Decryption	x									
Hashing		xw								
Acquiring a Table of Pointers to BCryptXXX Functions										
Algorithm Providers and										

Properties										
Key and Key-Pair Generation	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd
Key Entry and Output	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
Signing and Verification			x	x	x	x	x	x		
Secret Agreement and Key Derivation									x	x

7.3 Key Material

Each time an application links with BCRYPTPRIMITIVES.DLL, the DLL is instantiated and no keys exist within it. The user application is responsible for importing keys into BCRYPTPRIMITIVES.DLL or using BCRYPTPRIMITIVES.DLL's functions to generate keys.

7.4 Key Generation

BCRYPTPRIMITIVES.DLL can create and use keys for the following algorithms: RSA, DSA, DH, ECDH, ECDSA, RC2, RC4, DES, Triple-DES, AES, and HMAC.
Random keys can be generated by calling the BCryptGenerateSymmetricKey() and BCryptGenerateKeyPair() functions. Random data generated by the BCryptGenRandom() function is provided to BCryptGenerateSymmetricKey() function to generate symmetric keys. DES, Triple-DES, AES, RSA, ECDSA, DSA, DH, and ECDH keys and key-pairs are generated following the techniques given in section 5.2.

7.5 Key Establishment

BCRYPTPRIMITIVES.DLL can use FIPS approved Diffie-Hellman key agreement (DH), Elliptic Curve Diffie-Hellman key agreement (ECDH), RSA key transport and manual methods to establish keys.
BCRYPTPRIMITIVES.DLL can use the following FIPS approved key derivation functions (KDF) from the common secret that is established during the execution of DH and ECDH key agreement algorithms:

- BCRYPT_KDF_SP80056A_CONCAT. This KDF supports the Concatenation KDF as specified in SP 800-56A (Section 5.8.1).
- BCRYPT_KDF_HASH. This KDF supports FIPS approved SP800-56A (Section 5.8), X9.63, and X9.42 key derivation.
- BCRYPT_KDF_HMAC. This KDF supports the IPsec IKEv1 key derivation that is allowed in FIPS mode when used to establish keys for IKEv1 as specified in FIPS 140-2 Implementation Guidance 7.1.
- BCRYPT_KDF_TLS_PRF. This KDF supports the SSLv3.1 and TLSv1.0 key derivation that is allowed in FIPS mode when used to establish keys for SSLv3.1 or TLSv1.0 as specified in FIPS 140-2 Implementation Guidance 7.1.

7.6 Key Entry and Output

Keys can be both exported and imported out of and into BCRYPTPRIMITIVES.DLL via BCryptExportKey(), BCryptImportKey(), and BCryptImportKeyPair() functions.
Symmetric key entry and output can also be done by exchanging keys using the recipient's asymmetric public key via BCryptSecretAgreement() and BCryptDeriveKey() functions.

Exporting the RSA private key by supplying a blob type of BCRYPT_PRIVATE_KEY_BLOB, BCRYPT_RSAFULLPRIVATE_BLOB, or BCRYPT_RSAPRIVATE_BLOB to BCryptExportKey() is not allowed in FIPS mode.

7.7 Key Storage

BCRYPTPRIMITIVES.DLL does not provide persistent storage of keys.

7.8 Key Archival

BCRYPTPRIMITIVES.DLL does not directly archive cryptographic keys. The Authenticated User may choose to export a cryptographic key (cf. "Key Entry and Output" above), but management of the secure archival of that key is the responsibility of the user.

7.9 Key Zeroization

All keys are destroyed and their memory location zeroized when the operator calls BCryptDestroyKey() or BCryptDestroySecret() on that key handle.

8 Self-Tests

BCRYPTPRIMITIVES.DLL performs the following power-on (start up) self-tests when DllMain is called by the operating system.

- HMAC-SHA-1Known Answer Test
- SHA-256 and SHA-512 Known Answer Tests
- Triple-DES encrypt/decrypt ECB Known Answer Test
- AES-128 encrypt/decrypt ECB Known Answer Test
- AES-128 encrypt/decrypt CBC Known Answer Test
- AES-128 encrypt/decrypt CCM Known Answer Test
- AES-128 encrypt/decrypt GCM Known Answer Test
- DSA sign/verify test with 1024-bit key
- RSA sign and verify test with 2048-bit key
- DH secret agreement Known Answer Test with 1024-bit key
- ECDSA sign/verify test on P256 curve
- ECDH secret agreement Known Answer Test on P256 curve
- SP800-56A concatenation KDF Known Answer Tests
- FIPS 186-2 DSA random generator Known Answer Tests
- SP800-90 AES-256 based counter mode random generator Known Answer Tests (instantiate, generate and reseed)
- SP800-90 dual elliptic curve random generator Known Answer Tests (instantiate, generate and reseed)

BCRYPTPRIMITIVES.DLL performs pair-wise consistency checks upon each invocation of RSA, ECDH, DSA, and ECDSA key-pair generation and import as defined in FIPS 140-2. BCRYPTPRIMITIVES.DLL also performs a continuous RNG test on each of the implemented RNGs as defined in FIPS 140-2.

In all cases for any failure of a power-on (start up) self-test, BCRYPTPRIMITIVES.DLL DllMain fails to return the STATUS_SUCCESS status to the operating system. The only way to recover from the failure of a power-on (start up) self-test is to attempt to reload the BCRYPTPRIMITIVES.DLL, which will rerun the self-tests, and will only succeed if the self-tests passes.

9 Design Assurance

The BCRYPTPRIMITIVES.DLL crypto module is part of the overall Windows 7 operating system, which is a product family that has gone through and is continuously going through the Common Criteria Certification or equivalent under US NIAP CCEVS since Windows NT 3.5. The certification provides the necessary design assurance.
The BCRYPTPRIMITIVES.DLL is installed and started as part of the Windows 7 operating system.

10 Additional details

For the latest information on Windows 7, check out the Microsoft web site at http://www.microsoft.com.

CHANGE HISTORY			
AUTHOR	**DATE**	**VERSION**	**COMMENT**
Tolga Acar	6/7/2007	1.0	FIPS Approval Submission
Stefan Santesson	10/30/2007	1.1	Added technical updates related to SP1 and WS2K8
Stefan Santesson	2/15/2008	1.2	Merged changes resulting from Gold CMVP review
Vijay Bharadwaj	2/28/2008	1.3	Revisions for SP1 and WS08
Vijay Bharadwaj	5/5/2009	2.0	Windows 7 changes – moving from BCRYPT to BCRYPTPRIMITIVES

EXHIBIT K

TPM Key Attestation

05/31/2017 • 13 minutes to read • +7

In this article

> Applies To: Windows Server 2016, Windows Server 2012 R2, Windows Server 2012

Author: Justin Turner, Senior Support Escalation Engineer with the Windows group

> **Note**
>
> This content is written by a Microsoft customer support engineer, and is intended for experienced administrators and systems architects who are looking for deeper technical explanations of features and solutions in Windows Server 2012 R2 than topics on TechNet usually provide. However, it has not undergone the same editing passes, so some of the language may seem less polished than what is typically found on TechNet.

Overview

While support for TPM-protected keys has existed since Windows 8, there were no mechanisms for CAs to cryptographically attest that the certificate requester private key is actually protected by a Trusted Platform Module (TPM). This update enables a CA to perform that attestation and to reflect that attestation in the issued certificate.

> ⓘ **Note**
>
> This article assumes that the reader is familiar with certificate template concept (for reference, see **Certificate Templates**). It also assumes that the reader is familiar with how to configure enterprise CAs to issue certificates based on certificate templates (for reference, see **Checklist: Configure CAs to Issue and Manage Certificates**).

Terminology

Term	Definition
EK	Endorsement Key. This is an asymmetric key contained inside the TPM (injected at manufacturing time). The EK is unique for every TPM and can identify it. The EK cannot be changed or removed.
EKpub	Refers to public key of the EK.
EKPriv	Refers to private key of the EK.
EKCert	EK Certificate. A TPM manufacturer-issued certificate for EKPub. Not all TPMs have EKCert.
TPM	Trusted Platform Module. A TPM is designed to provide hardware-based security-related functions. A TPM chip is a secure crypto-processor that is designed to carry out cryptographic operations. The chip includes multiple physical security mechanisms to make it tamper resistant, and malicious software is unable to tamper with the security functions of the TPM.

Background

Beginning with Windows 8, a Trusted Platform Module (TPM) can be used to secure a certificate's private key. The Microsoft Platform Crypto Provider Key Storage Provider (KSP) enables this feature. There were two concerns with the implementation:

- There was no guarantee that a key is actually protected by a TPM (someone can easily spoof a software KSP as a TPM KSP with local administrator credentials).
- It was not possible to limit the list of TPMs that are allowed to protect enterprise issued certificates (in the event that the PKI Administrator wants to control the types of devices that can be used to obtain certificates in the environment).

TPM key attestation

TPM key attestation is the ability of the entity requesting a certificate to cryptographically prove to a CA that the RSA key in the certificate request is protected by either "a" or "the" TPM that the CA trusts. The TPM trust model is discussed more in the Deployment overview section later in this topic.

Why is TPM key attestation important?

A user certificate with a TPM-attested key provides higher security assurance, backed up by non-exportability, anti-hammering, and isolation of keys provided by the TPM.

With TPM key attestation, a new management paradigm is now possible: An administrator can define the set of devices that users can use to access corporate resources (for example, VPN or wireless access point) and have **strong** guarantees that no other devices can be used to access them. This new access control paradigm is **strong** because it is tied to a *hardware-bound* user identity, which is stronger than a software-based credential.

How does TPM key attestation work?

In general, TPM key attestation is based on the following pillars:

1. Every TPM ships with a unique asymmetric key, called the *Endorsement Key* (EK), burned by the manufacturer. We refer to the public portion of this key as *EKPub* and the associated private key as *EKPriv*. Some TPM chips also have an EK certificate that is issued by the manufacturer for the EKPub. We refer to this cert as

EKCert.

2. A CA establishes trust in the TPM either via EKPub or EKCert.

3. A user proves to the CA that the RSA key for which the certificate is being requested is cryptographically related to the EKPub and that the user owns the EKpriv.

4. The CA issues a certificate with a special issuance policy OID to denote that the key is now attested to be protected by a TPM.

Deployment overview

In this deployment, it is assumed that a Windows Server 2012 R2 enterprise CA is set up. Also, clients (Windows 8.1) are configured to enroll against that enterprise CA using certificate templates.

There are three steps to deploying TPM key attestation:

1. **Plan the TPM trust model:** The first step is to decide which TPM trust model to use. There are 3 supported ways for doing this:

   - **Trust based on user credential:** The enterprise CA trusts the user-provided EKPub as part of the certificate request and no validation is performed other than the user's domain credentials.

   - **Trust based on EKCert:** The enterprise CA validates the EKCert chain that is provided as part of the certificate request against an administrator-managed list of *acceptable EK cert chains*. The acceptable chains are defined per-manufacturer and are expressed via two custom certificate stores on the issuing CA (one store for the intermediate and one for root CA certificates). This trust mode means that **all** TPMs from a given manufacturer are trusted. Note that in this mode, TPMs in use in the environment must contain EKCerts.

   - **Trust based on EKPub:** The enterprise CA validates that the EKPub provided as part of the certificate request appears in an administrator-managed list of

allowed EKPubs. This list is expressed as a directory of files where the name of each file in this directory is the SHA-2 hash of the allowed EKPub. This option offers the highest assurance level but requires more administrative effort, because each device is individually identified. In this trust model, only the devices that have had their TPM's EKPub added to the allowed list of EKPubs are permitted to enroll for a TPM-attested certificate.

Depending on which method is used, the CA will apply a different issuance policy OID to the issued certificate. For more details about issuance policy OIDs, see the Issuance Policy OIDs table in the Configure a certificate template section in this topic.

Note that it is possible to choose a combination of TPM trust models. In this case, the CA will accept any of the attestation methods, and the issuance policy OIDs will reflect all attestation methods that succeed.

2. **Configure the certificate template:** Configuring the certificate template is described in the Deployment details section in this topic. This article does not cover how this certificate template is assigned to the enterprise CA or how enroll access is given to a group of users. For more information, see Checklist: Configure CAs to Issue and Manage Certificates.

3. **Configure the CA for the TPM trust model**

   a. **Trust based on user credential:** No specific configuration is required.

   b. **Trust based on EKCert:** The administrator must obtain the EKCert chain certificates from TPM manufacturers, and import them to two new certificate stores, created by the administrator, on the CA that perform TPM key attestation. For more information, see the CA configuration section in this topic.

   c. **Trust based on EKPub:** The administrator must obtain the EKPub for each device that will need TPM-attested certificates and add them to the list of allowed EKPubs. For more information, see the CA configuration section in this topic.

> ⓘ **Note**
>
> - This feature requires Windows 8.1/Windows Server 2012 R2.
> - TPM key attestation for third-party smart card KSPs is not supported. Microsoft Platform Crypto Provider KSP must be used.
> - TPM key attestation only works for RSA keys.
> - TPM key attestation is not supported for a standalone CA.
> - TPM key attestation does not support **non-persistent certificate processing**.

Deployment details

Configure a certificate template

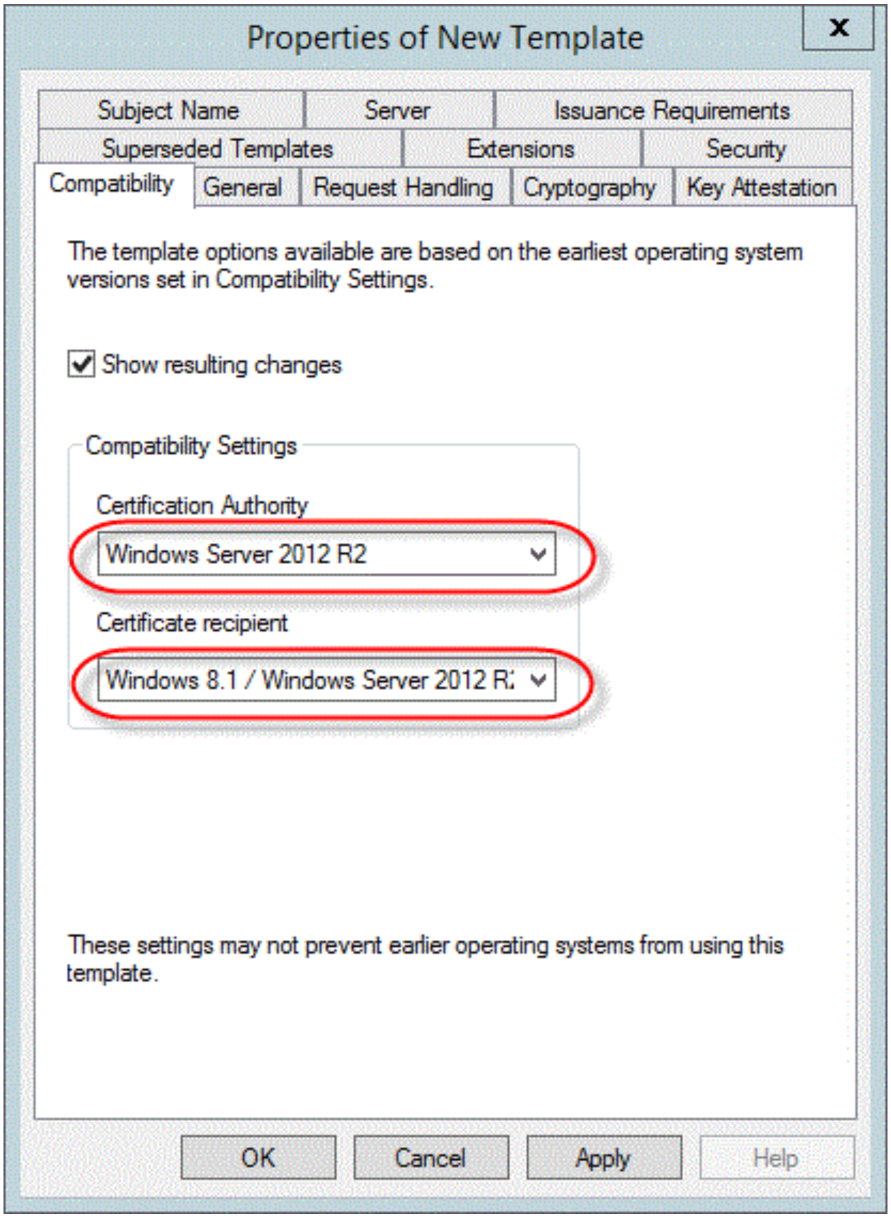
To configure the certificate template for TPM key attestation, do the following configuration steps:

1. **Compatibility** tab

   In the **Compatibility Settings** section:

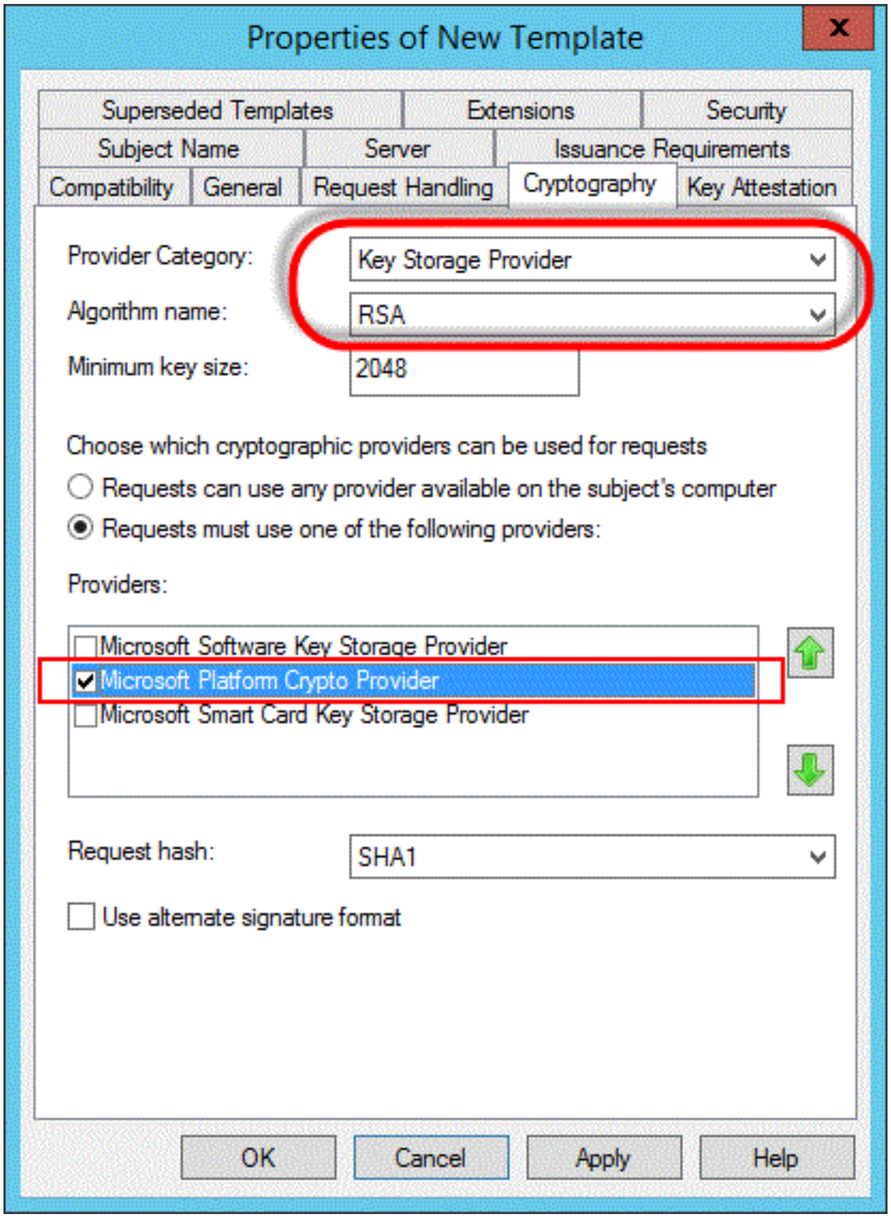
   - Ensure **Windows Server 2012 R2** is selected for the **Certification Authority**.

   - Ensure **Windows 8.1 / Windows Server 2012 R2** is selected for the **Certificate recipient**.



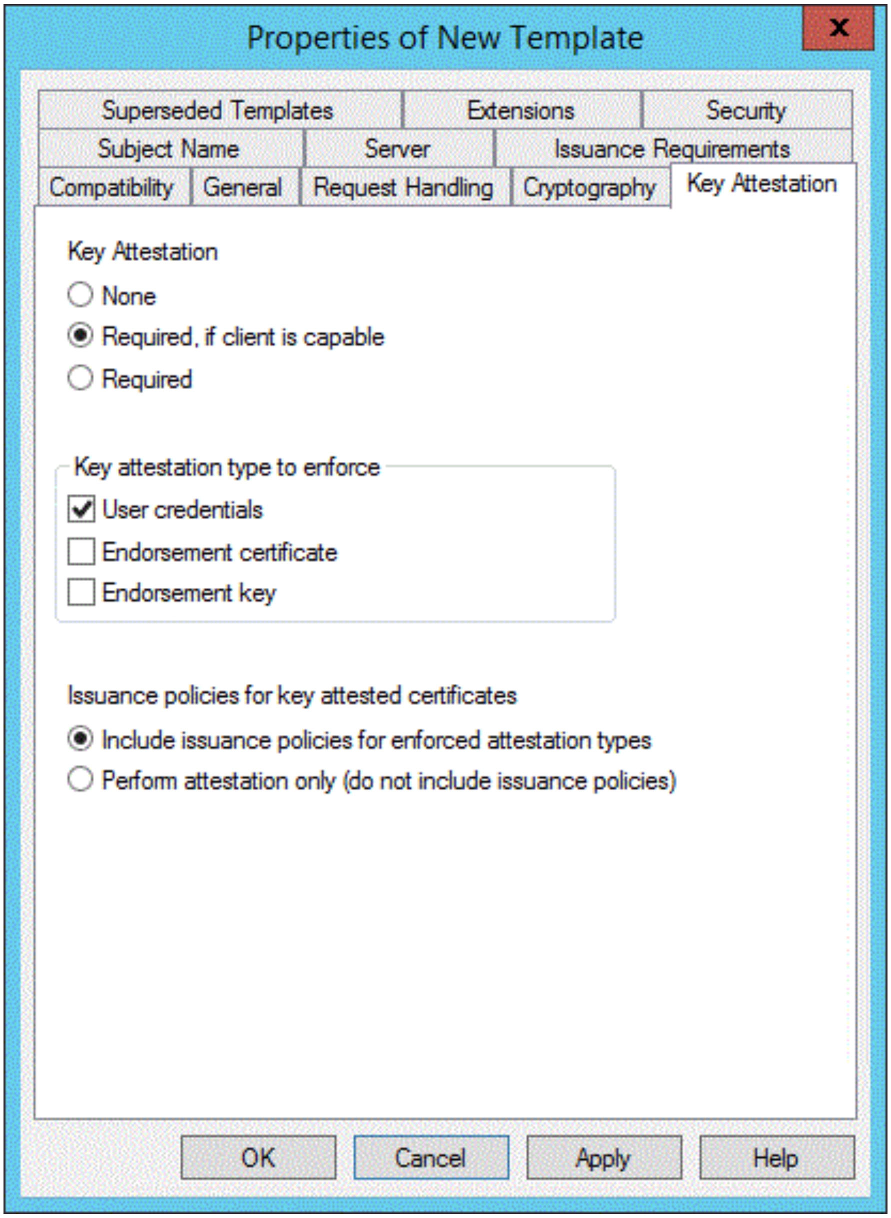
2. **Cryptography** tab

   Ensure **Key Storage Provider** is selected for the **Provider Category** and **RSA** is selected for the **Algorithm name**. Ensure **Requests must use one of the following providers** is selected and the **Microsoft Platform Crypto Provider** option is selected under **Providers**.

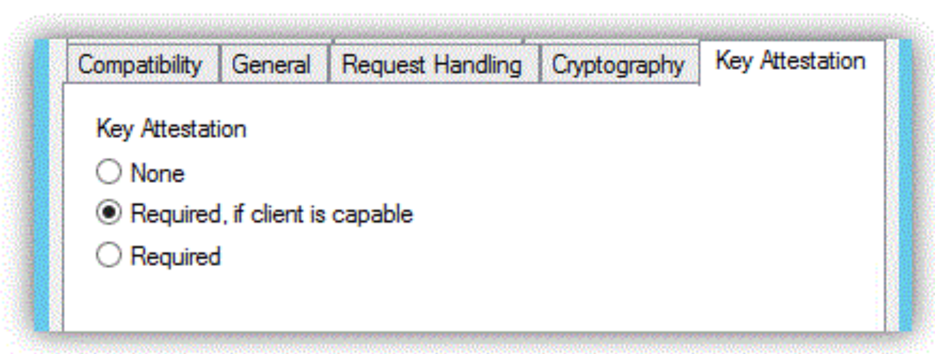


3. **Key Attestation** tab

   This is a new tab for Windows Server 2012 R2:



Choose an attestation mode from the three possible options.

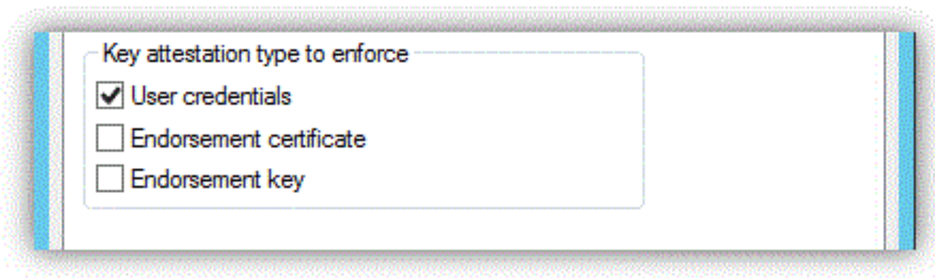


- **None:** Implies that key attestation must not be used

- **Required, if client is capable:** Allows users on a device that does not support TPM key attestation to continue enrolling for that certificate. Users who can perform attestation will be distinguished with a special issuance policy OID. Some devices might not be able to perform attestation because of an old TPM that does not support key attestation, or the device not having a TPM at all.

- **Required:** Client *must* perform TPM key attestation, otherwise the certificate request will fail.

Then choose the TPM trust model. There are again three options:



- **User credentials:** Allow an authenticating user to vouch for a valid TPM by specifying their domain credentials.

- **Endorsement certificate:** The EKCert of the device must validate through administrator-managed TPM intermediate CA certificates to an administrator-managed root CA certificate. If you choose this option, you must set up EKCA and EKRoot certificate stores on the issuing CA as described in the CA configuration section in this topic.

- **Endorsement Key:** The EKPub of the device must appear in the PKI administrator-managed list. This option offers the highest assurance level but requires more administrative effort. If you choose this option, you must set up an EKPub list on the issuing CA as described in the CA configuration section in this topic.

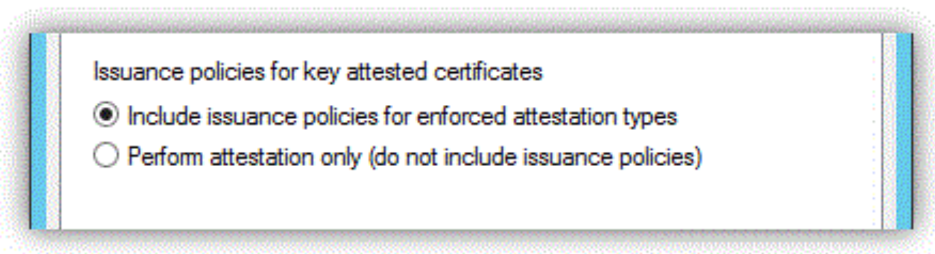
Finally, decide which issuance policy to show in the issued certificate. By default, each enforcement type has an associated object identifier (OID) that will be

inserted into the certificate if it passes that enforcement type, as described in the following table. Note that it is possible to choose a combination of enforcement methods. In this case, the CA will accept any of the attestation methods, and the issuance policy OID will reflect all attestation methods that succeeded.

Issuance Policy OIDs

OID	Key attestation type	Description	Assurance level
1.3.6.1.4.1.311.21.30	EK	"EK Verified": For administrator-managed list of EK	High
1.3.6.1.4.1.311.21.31	Endorsement certificate	"EK Certificate Verified": When EK certificate chain is validated	Medium
1.3.6.1.4.1.311.21.32	User credentials	"EK Trusted on Use": For user-attested EK	Low

The OIDs will be inserted into the issued certificate if **Include Issuance Policies** is selected (the default configuration).



Tip

One potential use of having the OID present in the certificate is to limit access to VPN or wireless networking to certain devices. For example, your access policy might allow connection (or access to a different VLAN) if OID

1.3.6.1.4.1.311.21.30 is present in the certificate. This allows you to limit access to devices whose TPM EK is present in the EKPUB list.

CA configuration

1. **Setup EKCA and EKROOT certificate stores on an issuing CA**

   If you chose **Endorsement Certificate** for the template settings, do the following configuration steps:

   a. Use Windows PowerShell to create two new certificate stores on the certification authority (CA) server that will perform TPM key attestation.

   b. Obtain the intermediate and root CA certificate(s) from manufacturer(s) that you want to allow in your enterprise environment. Those certificates must be imported into the previously-created certificate stores (EKCA and EKROOT) as appropriate.

   The following Windows PowerShell script performs both of these steps. In the following example, the TPM manufacturer Fabrikam has provided a root certificate *FabrikamRoot.cer* and an issuing CA certificate *Fabrikamca.cer*.

   PowerShell Copy

   ```
   PS C:>\cd cert:
   PS Cert:\>cd .\\LocalMachine
   PS Cert:\LocalMachine> new-item EKROOT
   PS Cert:\ LocalMachine> new-item EKCA
   PS Cert:\EKCA\copy FabrikamCa.cer .\EKCA
   PS Cert:\EKROOT\copy FabrikamRoot.cer .\EKROOT
   ```

2. **Setup EKPUB List if using EK attestation type**

   If you chose **Endorsement Key** in the template settings, the next configuration steps are to create and configure a folder on the issuing CA, containing 0-byte files, each named for the SHA-2 hash of an allowed EK. This folder serves as an

"allow list" of devices that are permitted to obtain TPM key-attested certificates. Because you must manually add the EKPUB for each and every device that requires an attested certificate, it provides the enterprise with a guarantee of the devices that are authorized to obtain TPM key attested certificates. Configuring a CA for this mode requires two steps:

a. **Create the EndorsementKeyListDirectories registry entry:** Use the Certutil command-line tool to configure the folder locations where trusted EKpubs are defined as described in the following table.

Operation	Command syntax
Add folder locations	certutil.exe -setreg CA\EndorsementKeyListDirectories +""
Remove folder locations	certutil.exe -setreg CA\EndorsementKeyListDirectories -""

The EndorsementKeyListDirectories in certutil command is a registry setting as described in the following table.

Value name	Type	Data
EndorsementKeyListDirectories	REG_MULTI_SZ	<LOCAL or UNC path to EKPUB allow list(s)> Example: *\\blueCA.contoso.com\ekpub* *\\bluecluster1.contoso.com\ekpub* D:\ekpub

HKLM\SYSTEM\CurrentControlSet\Services\CertSvc\Configuration\

EndorsementKeyListDirectories will contain a list of UNC or local file system paths, each pointing to a folder that the CA has Read access to. Each folder may contain zero or more allow list entries, where each entry is a file with a name

that is the SHA-2 hash of a trusted EKpub, with no file extension. Creating or editing this registry key configuration requires a restart of the CA, just like existing CA registry configuration settings. However, edits to the configuration setting will take effect immediately and will not require the CA to be restarted.

> ⓘ **Important**
>
> Secure the folders in the list from tampering and unauthorized access by configuring permissions so that only authorized administrators have Read and Write access. The computer account of the CA requires Read access only.

b. **Populate the EKPUB list:** Use the following Windows PowerShell cmdlet to obtain the public key hash of the TPM EK by using Windows PowerShell on each device and then send this public key hash to the CA and store it on the EKPubList folder.

PowerShell | Copy

```
PS C:>\$a=Get-TpmEndorsementKeyInfo -hashalgorithm sha256
PS C:>$b=new-item $a.PublicKeyHash -ItemType file
```

Troubleshooting

Key attestation fields are unavailable on a certificate template

The Key Attestation fields are not available if the template settings do not meet the requirements for attestation. Common reasons are:

1. The compatibility settings are not configured correctly. Make sure that they are configured as follows:

   a. **Certification Authority**: **Windows Server 2012 R2**

   b. **Certificate Recipient**: **Windows 8.1/Windows Server 2012 R2**

2. The cryptography settings are not configured correctly. Make sure that they are configured as follows:

   a. **Provider Category**: **Key Storage Provider**

   b. **Algorithm Name**: **RSA**

   c. **Providers**: **Microsoft Platform Crypto Provider**

3. The request handling settings are not configured correctly. Make sure that they are configured as follows:

   a. The **Allow private key to be exported** option must not be selected.

   b. The **Archive subject's encryption private key** option must not be selected.

Verification of TPM device for attestation

Use the Windows PowerShell cmdlet, **Confirm-CAEndorsementKeyInfo**, to verify that a specific TPM device is trusted for attestation by CAs. There are two options: one for verifying the EKCert, and the other for verifying an EKPub. The cmdlet is either run locally on a CA, or on remote CAs by using Windows PowerShell remoting.

1. For verifying trust on an EKPub, do the following two steps:

   a. **Extract the EKPub from the client computer:** The EKPub can be extracted from a client computer via **Get-TpmEndorsementKeyInfo**. From an elevated command prompt, run the following:

   Copy

   ```
   PS C:>\$a=Get-TpmEndorsementKeyInfo -hashalgorithm sha256
   ```

b. **Verify trust on an EKCert on a CA computer:** Copy the extracted string (the SHA-2 hash of the EKPub) to the server (for example, via email) and pass it to the Confirm-CAEndorsementKeyInfo cmdlet. Note that this parameter must be 64 characters.

Copy

```
Confirm-CAEndorsementKeyInfo [-PublicKeyHash] <string>
```

2. For verifying trust on an EKCert, do the following two steps:

a. **Extract the EKCert from the client computer:** The EKCert can be extracted from a client computer via **Get-TpmEndorsementKeyInfo**. From an elevated command prompt, run the following:

Copy

```
PS C:>\$a=Get-TpmEndorsementKeyInfo
PS C:>\$a.manufacturerCertificates|Export-Certificate -filepath
c:\myEkcert.cer
```

b. **Verify trust on an EKCert on a CA computer:** Copy the extracted EKCert (EkCert.cer) to the CA (for example, via email or xcopy). As an example, if you copy the certificate file the "c:\diagnose" folder on the CA server, run the following to finish verification:

Copy

```
PS C:>new-object System.Security.Cryptography.X509Certifi-
cates.X509Certificate2 "c:\diagnose\myEKcert.cer" | Confirm-
CAEndorsementKeyInfo
```

See Also

Trusted Platform Module Technology Overview External Resource: Trusted Platform

Module

Is this page helpful?

Yes No

EXHIBIT L

BCryptOpenAlgorithmProvider function (bcrypt.h)

12/05/2018 • 3 minutes to read

In this article

The **BCryptOpenAlgorithmProvider** function loads and initializes a CNG provider.

Syntax

C++ Copy

```
NTSTATUS BCryptOpenAlgorithmProvider(
  BCRYPT_ALG_HANDLE *phAlgorithm,
  LPCWSTR           pszAlgId,
  LPCWSTR           pszImplementation,
  ULONG             dwFlags
);
```

Parameters

`phAlgorithm`

A pointer to a **BCRYPT_ALG_HANDLE** variable that receives the CNG provider handle.

When you have finished using this handle, release it by passing it to the BCryptCloseAlgorithmProvider function.

`pszAlgId`

A pointer to a null-terminated Unicode string that identifies the requested cryptographic algorithm. This can be one of the standard CNG Algorithm Identifiers or the identifier for another registered algorithm.

`pszImplementation`

A pointer to a null-terminated Unicode string that identifies the specific provider to load. This is the registered alias of the cryptographic primitive provider. This parameter is optional and can be **NULL** if it is not needed. If this parameter is **NULL**, the default provider for the specified algorithm will be loaded.

> **Note** If the *pszImplementation* parameter value is **NULL**, CNG attempts to open each registered provider, in order of priority, for the algorithm specified by the *pszAlgId* parameter and returns the handle of the first provider that is successfully opened. For the lifetime of the handle, any BCrypt*** cryptographic APIs will use the provider that was successfully opened.

Windows Server 2008 and Windows Vista: CNG attempts to fall back to the Microsoft CNG provider.

The following are the predefined provider names.

Value	Meaning
MS_PRIMITIVE_PROVIDER "Microsoft Primitive Provider"	Identifies the basic Microsoft CNG provider.
MS_PLATFORM_CRYPTO_PROVIDER L"Microsoft Platform Crypto Provider"	Identifies the TPM key storage provider that is provided by Microsoft.

`dwFlags`

Flags that modify the behavior of the function. This can be zero or a combination of one or more of the following values.

Value	Meaning
BCRYPT_ALG_HANDLE_HMAC_FLAG	The provider will perform the Hash-Based Message Authentication Code (HMAC) algorithm with the specified hash algorithm. This flag is only used by hash algorithm providers.
BCRYPT_PROV_DISPATCH	Loads the provider into the nonpaged memory pool. If this flag is not present, the provider is loaded into the paged memory pool. When this flag is specified, the handle returned must not be closed before all dependent objects have been freed. **Note** This flag is only supported in kernel mode and allows subsequent operations on the provider to be processed at the Dispatch level. If the provider does not support being called at dispatch level, then it will return an error when opened using this flag. **Windows Server 2008 and Windows Vista:** This flag is only supported by the Microsoft algorithm providers and only for hashing algorithms and symmetric key cryptographic algorithms.
BCRYPT_HASH_REUSABLE_FLAG	Creates a reusable hashing object. The object can be used for a new hashing operation immediately after calling BCryptFinishHash. For more information, see Creating a Hash with CNG. **Windows Server 2008 R2, Windows 7, Windows Server 2008 and Windows Vista:** This flag is not

supported.

Return value

Returns a status code that indicates the success or failure of the function.

Possible return codes include, but are not limited to, the following.

Return code	Description
STATUS_SUCCESS	The function was successful.
STATUS_NOT_FOUND	No provider was found for the specified algorithm ID.
STATUS_INVALID_PARAMETER	One or more parameters are not valid.
STATUS_NO_MEMORY	A memory allocation failure occurred.

Remarks

Because of the number and type of operations that are required to find, load, and initialize an algorithm provider, the **BCryptOpenAlgorithmProvider** function is a relatively time intensive function. Because of this, we recommend that you cache any algorithm provider handles that you will use more than once, rather than opening and closing the algorithm providers over and over.

BCryptOpenAlgorithmProvider can be called either from user mode or kernel mode. Kernel mode callers must be executing at **PASSIVE_LEVEL** IRQL.

To call this function in kernel mode, use Cng.lib, which is part of the Driver Development Kit (DDK). **Windows Server 2008 and Windows Vista:** To call this function in kernel mode, use Ksecdd.lib.

Starting in Windows 10, CNG no longer follows every update to the cryptography

configuration. Certain changes, like adding a new default provider or changing the preference order of algorithm providers, may require a reboot. Because of this, you should reboot before calling **BCryptOpenAlgorithmProvider** with any newly configured provider.

Requirements

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	bcrypt.h
Library	Bcrypt.lib
DLL	Bcrypt.dll

See also

BCryptCloseAlgorithmProvider

Is this page helpful?

Yes No

EXHIBIT M

Using the Windows 8 Platform Crypto Provider and Associated TPM Functionality

Functionality, Usage Models, and Reference Implementation

White Paper

Stefan Thom, stefanth@microsoft.com
Jork Loeser, jloeser@microsoft.com
Ron Aigner, raigner@microsoft.com
Paul England, pengland@microsoft.com
Rob Spiger, rspiger@microsoft.com
Jim Morgan, jimorg@microsoft.com

Version 1.0

Table of Contents

Introduction to Attestation and PCP-Kit

This paper describes how a software provider can use the Microsoft® Windows® operating system and the Trusted Platform Module (TPM) to provide more reliable reporting of the health or policy compliance of computer systems and strong attestation of key origin and key properties. It also describes core operating system (OS) features for creating and using TPM keys that are bound to the physical machine, and how provisioning and other actions are performed. Finally, this paper describes a package of sample code and utilities called the Platform Configuration Provider Helper-Kit (PCP-Kit).

A Trusted Platform Module forms the low-level protected Root of Trust for Windows. The TPM can be a discrete cryptographic processor that is physically attached to the motherboard or may be an integrated implementation that provides similar security properties. One of the key capabilities of the TPM is to allow the authoritative reporting of the software running on the platform. This capability is called TPM-based attestation.

Many enterprises check software state and OS policy compliance before allowing computers to access corporate network resources. The goal of these checks is to ensure that the OS is properly patched, the OS configuration meets company policy, and that antivirus software is up-to-date. Unfortunately, in today's systems, this reporting is not very reliable because a genuine statement of system health can be spoofed by a rootkit or other malware. Attestation can provide a much more reliable anchor of trust for all online activities. Attestation uses the TPM to provide a cryptographically strong description of the platform configuration: With attestation, malware has nowhere to hide.

One of the central challenges with making attestation practical is developing policies for operating system components and settings that are useful. In particular, if reporting is too detailed (for example, the system provides details about every OS component and every security-sensitive OS setting), then it is hard to interpret which states are safe and which are not.

The approach built into Windows is to measure core OS components (which seldom change) and a specially vetted driver that is responsible for checking that the system meets policy. This specially vetted driver—called the Early Launch Anti-Malware (ELAM) driver—commonly checks for malware. .

Additionally, antivirus software is typically structured to include a core-detection engine (which also seldom changes) and a virus definition file that changes relatively often. Platform trust depends on both these components, so such configuration files are also included in the platform measurements.

The system firmware and the Windows OS record integrity measurements during boot in the TPM and maintain a log of the measurements in memory. For attestation, third-party software must be used to interpret this information to make security decisions. Typically, the solutions involve both client-side code and server/cloud-side software and services. On the client side, antivirus software will be enhanced to use attestation to report system configuration. On the server side, system health-monitoring applications or network-access control services will query the health of clients that request access. The server systems then receive and interpret attestation reports, and will grant network access (or raise an alert) based on the health reports received.

This paper is distributed with sample code and a utility called PCPTool. The tool and the provided sources are designed to help AV-system vendors make use of the attestation facilities in Windows.

Scope

The scope of this paper is an introduction to the Windows 8 capabilities, application programming interfaces (APIs), and properties around TPM 1.2 and 2.0 and a compilation of information required to design a solution. On a more practical level, the PCP-Kit is provided to allow direct experimentation and evaluation on a live Windows 8 operating system.

PCP-Kit and TPM Versions

At the time of writing, most TPMs are based on the TPM 1.2 specification published by the Trusted Computing Group (TCG). The new version of the TPM is referred to by Microsoft as "TPM 2.0"; however, the TCG refers to it as "TPM, family 2.0." The TPM 1.2 and TPM 2.0 devices are not compatible; however, Windows 8 supports both classes of devices, and some TPM-based services that are built into Windows 8 use an adaptation layer that transparently uses whatever device is present.

Windows supports attestation by allowing applications direct access to the underlying TPM so that the applications themselves can provide attestation services. This places the onus on the application to understand two complex devices. To lessen that burden, the PCP-Kit contains sample library code that provides a common hardware abstraction layer, as well as raising the functional level of the platform services provided.

We strongly suggest that solution providers obtain third-party TPM-library solutions or use or adapt the PCP-Kit library code to minimize application complexity.

Key Concepts Used in This Paper

Attestation is a complex subject. Solution providers can avoid some of the complexity if they use the code and utilities in the PCP-Kit, but some concepts and jargon must be explained so they can understand the facilities described in this paper. This section sketches some core TPM concepts, and provides a high-level overview of the Windows OS and PCP-Kit capabilities.

The Endorsement Key or EK

The TPM has an embedded unique cryptographic key called the *Endorsement Key,* or *EK*. The EK may be created during manufacturing, and for most TPMs the EK will remain in the TPM for the life of the device.

The EK is designed to provide a reliable cryptographic identifier for the platform. An enterprise might maintain a database of the Endorsement Keys belonging to the TPMs of all of the PCs in their enterprise, or a data center fabric controller might have a database of the TPMs in all of the blades. On Windows you can use the NCrypt provider described in the section "Platform Crypto Provider in Windows 8" to read the public part of the EK.

All TPMs compatible with Windows use a 2,048-bit RSA Endorsement Key. This TPM Endorsement Key is often accompanied by one or two *digital certificates*. One certificate is produced by the TPM manufacturer and is called the *endorsement certificate.* Another is produced by the platform builder and is called the *platform certificate* to indicate that a particular TPM is integrated with a certain machine. The owner of the machine may produce a third EK certificate to indicate ownership of the device. Such a certificate would be an *enterprise certificate*. On Windows you can use the NCrypt provider described in

the section "Platform Crypto Provider in Windows 8" to access the EK certificate store that contains all certificates for the EK on the platform. The administrator may add or remove certificates from this store.

The Endorsement Key can be used for direct machine authentication using the TPM functions TPM_ActivateIdentity (TPM 1.2) or TPM2_ActivateCredential (TPM 2.0). These functions are based on public key decryption using the Endorsement Key. However, the TPM allows secondary keys to be created that serve as delegates for the EK. These delegates are separate keys called *Attestation Identity Keys* or *AIKs*, and are described in the next section. Individual AIKs can be used with different services to avoid correlation based on the unique EK.

Attestation Identity Keys (AIKs)

Attestation Identity Keys are delegates for the EK that may be cryptographically certified by a third party called an *identity service provider* or *ISP*. Windows creates AIKs in the TPM that are 2,048-bit RSA signing keys. It is the job of the identity service provider to cryptographically establish that it is communicating with a real TPM (or a particular TPM, or a TPM in a group) and that the TPM has created a new AIK. Once the identity service provider has established these facts, it can create a new certificate for the new AIK-based identity.

The AIK certificate can contain whatever information the ISP thinks is appropriate. If the AIK is strictly to be used inside an enterprise it might be sufficient to simply record the public parts of the AIK in a database and no certificate for it is required. At the other end of the spectrum, the digital certificate might only record that the AIK belongs to a legitimate TPM. This might be appropriate for peer-to-peer trust on the Internet.

The key benefit of an AIK is that the server-side component to an attestation solution can verify that the measurements actually came from the specific client it intends to verify. This is why each server-side solution looking at data from a client system may want to establish its own AIK for a client, or use an AIK provisioned by an identity service provider it trusts to establish a trust association with the client to be verified.

The PCP-Kit contains sample code and tools for both client- and server-side AIK creation and certification. These functions are described more in the sections "Platform Crypto Provider in Windows 8" and "Attestation API Reference Implementation."

The AIKs directly support attestation because they can be used to sign the platform configuration. The way that Windows 8 records platform configurations is described in the next section.

It is a best practice from a privacy perspective for solutions to leverage an identity service provider because it can provide a level of anonymity for clients.

Platform Configuration and Platform Configuration Registers (or PCRs)

The TPM contains a set of registers that are designed to provide a cryptographic representation of the software and state of the system that booted. These registers are called *platform configuration registers* or *PCRs*. PCRs are set to zero when the platform is booted, and is the job of the software that boots the platform to measure software in the boot chain and to record the measurements in the PCRs. Typically, boot components take the hash of the next component that is to be run and record the hash measurements in the PCRs. The initial component that starts the measurement chain is implicitly trusted. It is called the

Core Root of Trust for Measurement. Platform manufacturers are required to have a secure update process for the Core Root of Trust for Measurement or not permit updates to it.

The PCRs record a cumulative hash of the components that have been measured. The value in a PCR on its own is hard to interpret (it is just a hash value), but platforms typically keep a log with details of the software and configurations that have been recorded, and the PCRs merely ensure that the log has not been tampered with. The logs are described in more detail in the section "Windows Boot Configuration Log" and also in the TCG specifications[1].

In Windows, the OS boot components record the OS loader, the OS kernel and all boot-start drivers, and specially signed Early Launch Anti-Malware drivers, as well as any necessary configuration files. This means that PCRs can report both the precise details of the OS that is running, the precise ELAM driver that has been loaded and initialized, and the policy that is being checked or enforced by the ELAM driver (for instance, a hash value that represents a dated virus definition file). The ELAM driver is a small driver with a small policy database that has a very narrow scope, focused on drivers that are loaded early at system launch. The policy database is stored in a new registry hive that is also measured to the TPM, to record the operational parameters of the ELAM driver.

The ELAM driver is initialized by the OS and is responsible for ensuring that later-loading components and configurations are within its policy until the regular AV driver is loaded and initialized. If the ELAM driver detects a policy violation (a known rootkit, for example), it may invalidate the PCRs that indicated the system was in a good state. This is done with a new OS call named Tbsi_Revoke_Attestation(), and is described in more detail in the section "Invalidating the System Trust State." After the regular full-scale AV driver is initialized and running, the ELAM driver and the ELAM hive will be unloaded.

This relatively simple model is made somewhat more complex by system hibernate and resume cycles. This is described in more detail in the section "Platform Trust Considerations across Hibernation and Resume."

Attestation

This section describes how PCRs (that contain system configuration data) and AIKs (that can report platform state) are used for configuration reporting.

As already described, the platform firmware and the operating system – in conjunction with the ELAM driver – will ensure that the platform configuration registers and the associated TCG logs are an accurate representation of the platform state.

Before the platform can *report* its configuration using the TPM attestation functions, an AIK must be created or provisioned in conjunction with a third party to achieve strong trust in the key. Clients and servers can use the PCPTool command-line utility or the PCP-Kit TpmAttPubKeyFromIdBinding and TpmAttGenerateActivation library functions to perform these actions (as described in the section "Creating Attestation Identity Keys (AIKs) and Forming Remote Trust").

Once provisioned, the AIK can be used in conjunction with the PCP-Kit sample/library code routines such as TpmAttCreateAttestationfromLog to report platform configuration. If the AIK is placed at a defined location in the registry, the OS will also create a signature over the platform log state (and a monotonic counter value) at each boot.

---

[1] Please refer to www.trustedcomputinggroup.org/developers/pc_client

Typically, the logs and statements of platform health are interpreted on servers. Checking that a TPM attestation and the associated log are valid takes several steps.

First, the server must check that the reports are signed by trustworthy AIKs. This might be done by checking that the public part of the AIK is listed in a database of assets, or perhaps that a certificate has been checked.

Once the key has been checked, the signed attestation (a quote structure) should be checked to see whether it is a valid signature over PCR values. Server code can use the TpmAttValidateKeyAttestation library routine, or the PCPTool utility.

Next the logs should be checked to ensure that they match the PCR values reported.

Finally, the logs themselves should be examined to see whether they represent known or valid security configurations. For instance, a simple check might be to see whether the measured early OS components are known good, that the ELAM driver is as expected, and that the ELAM-driver policy file is up-to-date (these checks are beyond the scope of this paper).

If all of these checks succeed, an attestation statement can be issued that later can be used to determine whether or not the client should be granted access to a resource.

Acronyms and Abbreviations

ACPI – Advanced Configuration and Power Interface

AIK – Attestation Identity Key. A TPM key that serves as an identity for the computer platform.

BCrypt – Algorithm provider infrastructure in CNG

CA – Certificate Authority

CAPI – Crypto API in Windows

Cert – Short for certificate

CNG – Crypto Next Generation API in Windows. This API is the successor of Crypto API (CAPI).

EK – Endorsement Key. A TPM key that is a cryptographic identifier for the TPM.

ELAM – Early Launch Anti-Malware. A driver that is loaded early by Windows and is responsible for checking and enforcing the early boot security policy.

EPS – Endorsement Primary Seed. A value from which a TPM 2.0 Endorsement Key is generated.

Key_{PUB} – Public portion of an asymmetric key

KSP – Key Storage Provider, a service of CNG that is accessed with the NCrypt APIs

NCrypt – Key storage provider infrastructure in CNG

PCP – Platform Crypto Provider

PCPKSP – Platform Crypto Provider Key Storage Provider

PPS – Platform Primary Seed. A value from which a TPM 2.0 platform key is generated.

SPS – Storage Primary Seed. A value from which a TPM 2.0 SRK is generated.

SRK – Storage Root Key

TBS – TPM Base Services

TCG – The Trusted Computing Group. The organization that is responsible for the TPM specification and related standards.

TPM – Trusted Platform Module

WBCL – Windows Boot Configuration Log

VSC – virtual smart card

TPM Provisioning and Management

The Windows 8 operating system is designed to automatically provision the TPM. This is in contrast to earlier versions of Windows where the end user had to provide explicit administrative actions. Once provisioned, Windows retains enough information to enable advanced TPM scenarios for itself or third-party applications. This also contrasts with earlier Windows versions where an application or TCG Software Stack (TSS) needed the TPM owner authorization value to be explicitly specified in order to perform advanced TPM scenarios.

Pre-Windows 8 Architecture for TPM 1.2 Provisioning

Most TPM systems in the marketplace in 2011 ship with the TPM 1.2 *deactivated* and *disabled*. When the TPM is deactivated it cannot be used for any interesting scenarios. We call the act of making the TPM ready to use *provisioning.* On current systems, the computer-platform firmware typically requires someone physically present (sitting at the keyboard) to approve changing the TPM state during the boot process.

This architecture has proven very complex for owners and system administrators. Few end users will understand unexpected BIOS/firmware prompts and questions, and system administrators dislike the fact that they cannot really do remote administration.

Some OEMs have provided tools and utilities to simplify remote administration, but these are hard to deploy in heterogeneous environments.

Windows 8 drastically simplifies the provisioning of TPM systems, as described in the next few sections.

Auto-Provisioning

Windows 8 will auto-provision the TPM so it is ready for use when applications want to use it. Windows 8 will not auto-provision the TPM if the provisioning process requires administrator interaction to complete the process.

Auto-provisioning actions will happen shortly after a full boot completes on a system with Windows 8. This means auto-provisioning usually won't occur after resuming from sleep or hibernation, including when hibernation technology hibernates the system core. (Generally, choosing to restart the system will cause a full boot.)

If the auto-provisioning actions find the TPM is deactivated or disabled, the provisioning actions will check the system capabilities to determine if it implements the TCG Physical Presence Interface Specification 1.2 and then if the system has the NoPPIProvision flag set to TRUE. If the flag is TRUE, it means the OS can initiate enabling and activating the TPM without any user involvement. The OS will request that the firmware enable and activate the TPM on the next boot. Upon the next restart of the OS, the TPM will be enabled and activated.

If the auto-provisioning actions find the TPM is enabled, activated, and ready to have ownership taken, the OS will take ownership of the TPM and perform a series of actions to set it up for use.

Scenarios that could cause Windows to not complete auto-provisioning are:

- The TPM is disabled or deactivated and the system does not implement the TCG Physical Presence Interface Specification 1.2 or does not have the NoPPIProvision flag set to TRUE.
- Ownership of the TPM was previously taken. This can happen if Windows is erased and reinstalled without clearing the TPM. It can also happen with an upgrade from a machine running Windows 7 or Windows Vista® that is using BitLocker® Drive Encryption.
- Additional Windows actions associated with provisioning the TPM cannot be completed. An example would be if Group Policy is configured to back up the TPM owner authorization value to Active Directory® service, but no connection to Active Directory is available.

Provisioning Through the UI

To manually provision the TPM, an administrator or an application can use the TPM Provisioning Wizard. The wizard is named tpminit.exe and may be run as an executable (Start->Run->tpminit.exe). The wizard will interactively walk the administrator through the process of provisioning the TPM.

TPM State in the OS

The TPM state in the Windows 8 operating system has been simplified for the user interface. The TPM may show one of the following statuses:

- "Ready for use" – The TPM and system are fully provisioned for TPM-related uses.
- "Ready for use, with Reduced Functionality" – Ownership of the TPM has been taken, but other system or TPM configuration actions are not completed.
- "Not Ready" – The TPM and system are not ready for use.

Provisioning with WMI

Several new Windows 8 Windows Management Instrumentation (WMI) methods help with TPM management. The methods IsReady and IsReadyInformation are useful for determining whether the TPM is fully ready for use or why it isn't. Another method, Provision, encapsulates the complex logic Windows 8 uses to provision the TPM into a single method for third- parties to use if they want to present their own provisioning user interface.

IsReady

This method determines whether the TPM and system are fully provisioned and ready for TPM use. The TPM may still be useful for some scenarios even if this method returns FALSE.

```
uint32 IsReady(

      [OUT] boolean IsReady

);
```

Parameters:
`[OUT] boolean IsReady` – Set to TRUE if the TPM and system are fully provisioned for TPM use.

Return value:
If the function succeeds, the function returns S_OK (0).

Remarks:
The return value indicates more than just TPM state. For example, if the system is configured to back up the TPM owner authorization value to Active Directory and the task has not been completed, IsReady will be set to FALSE.

This method is expensive to run because it performs many checks and does not cache its return values. It is recommended applications use this method only when necessary (for example, for provisioning and for troubleshooting).

IsReadyInformation

This method returns the status of the TPM and system and whether or not the TPM is provisioned and ready for use.

```
uint32 IsReadyInformation(

      [OUT] boolean IsReady,

      [OUT] uint32 Information

);
```

Parameters:
`[OUT] boolean IsReady` – Set to TRUE if the TPM and system are fully provisioned for TPM use.

`[OUT] uint32 Information` – Returns a bitmask of as much information as is available of what is needed to fully provision the TPM.

The Information bitmask may consist of the following values:

Symbol	Value	Description
INFORMATION_SHUTDOWN	0x00000002	Platform restart is required (shutdown).
INFORMATION_REBOOT	0x00000004	Platform restart is required (reboot).
INFORMATION_TPM_FORCE_CLEAR	0x00000008	The TPM is already owned. Either the TPM needs to be cleared or the TPM owner authorization value needs to be imported.
INFORMATION_PHYSICAL_PRESENCE	0x00000010	Physical Presence is required to provision the TPM.
INFORMATION_TPM_ACTIVATE	0x00000020	The TPM is disabled or deactivated.
INFORMATION_TPM_TAKE_OWNERSHIP	0x00000040	Ownership was taken.
INFORMATION_TPM_CREATE_EK	0x00000080	An Endorsement Key was created.
INFORMATION_TPM_OWNERAUTH	0x00000100	The TPM owner authorization is not properly stored in the registry.
INFORMATION_TPM_SRK_AUTH	0x00000200	The Storage Root Key authorization value is not all zeros.
INFORMATION_TPM_DISABLE_OWNER_CLEAR	0x00000400	Windows is configured to disable clearing of the TPM with the TPM owner authorization value and the TPM has not been configured to prevent the clearing.
INFORMATION_TPM_SRKPUB	0x00000800	The Windows registry information about the TPM's Storage Root Key does not match the

		TPM Storage Root Key.
INFORMATION_TPM_READ_SRKPUB	0x00001000	The TPM permanent flag to allow reading of the Storage Root Key public value is not set.
INFORMATION_TPM_BOOT_COUNTER	0x00002000	The monotonic counter incremented during boot has not been created.
INFORMATION_TPM_AD_BACKUP	0x00004000	The TPM's owner authorization has not been backed up to Active Directory.
INFORMATION_TPM_AD_BACKUP_PHASE_I	0x00008000	The first portion of the TPM owner authorization information storage in Active Directory is in progress.
INFORMATION_TPM_AD_BACKUP_PHASE_II	0x00010000	The second portion of the TPM owner authorization information storage in Active Directory is in progress.
INFORMATION_LEGACY_CONFIGURATION	0x00020000	Windows Group Policy is configured to not store any TPM owner authorization so the TPM cannot be fully ready.
TBD	0x00040000	The EK certificate was not read from the TPM non-volatile (NV) RAM and stored in the registry. (See the NCrypt Property NCRYPT_PCP_EKCERT_PROPERTY for more information.)
INFORMATION_TCG_EVENT_LOG	0x00080000	The TCG event log is empty or cannot be read. (This is usually a problem with the system firmware, not TPM state.)
INFORMATION_NOT_REDUCED	0x00100000	The TPM is not owned or otherwise not ready for use by BitLocker.
INFORMATION_GENERIC_ERROR	0x00200000	A generic error occurred.

Return value:
If the function succeeds, the function returns S_OK (0).

Remarks:
This method is expensive to run because it performs many checks and does not cache its return values. It is recommended applications use this method only when necessary (for example, for provisioning and for troubleshooting).

Provision

This method provisions the TPM and the system for use.

```
uint32 Provision(

      [IN] boolean ForceClear_Allowed,

      [IN] Boolean PhysicalPresencePrompts_Allowed,

      [OUT] uint32 Information

);
```

Parameters:

`[IN] boolean ForceClear_Allowed` – When set to TRUE, the method may request Physical Presence operations to clear the TPM. If set to FALSE, the method will not request a Physical Presence operation to clear the TPM.

`[IN] Boolean PhysicalPresencePrompts_Allowed` – When set to TRUE, the method may request Physical Presence operations that require user involvement during the boot process to confirm the TPM state change.

`[OUT] uint32 Information` – Returns a bitmask of as much information as is available of what is needed to fully provision the TPM. Mask values like INFORMATION_REBOOT indicate the method call should initiate a reboot to move the provisioning process forward.

The Information bitmask may consist of the following values:

Symbol	Value	Description
INFORMATION_SHUTDOWN	0x00000002	Platform restart is required (shutdown).
INFORMATION_REBOOT	0x00000004	Platform restart is required (reboot).
INFORMATION_TPM_FORCE_CLEAR	0x00000008	The TPM is already owned. Either the TPM needs to be cleared or the TPM owner authorization value needs to be imported.
INFORMATION_PHYSICAL_PRESENCE	0x00000010	Physical Presence is required to provision the TPM.
INFORMATION_TPM_ACTIVATE	0x00000020	The TPM is disabled or deactivated.
INFORMATION_TPM_TAKE_OWNERSHIP	0x00000040	Ownership was taken.
INFORMATION_TPM_CREATE_EK	0x00000080	An Endorsement Key was created.
INFORMATION_TPM_OWNERAUTH	0x00000100	The TPM owner authorization is not properly stored in the registry.
INFORMATION_TPM_SRK_AUTH	0x00000200	The Storage Root Key authorization value is not all zeros.
INFORMATION_TPM_DISABLE_OWNER_CLEAR	0x00000400	Windows is configured to disable clearing of the TPM with the TPM owner authorization value and the TPM has not been configured to prevent the clearing.
INFORMATION_TPM_SRKPUB	0x00000800	The Windows registry information about the TPM's Storage Root Key does not match the TPM Storage Root Key.
INFORMATION_TPM_READ_SRKPUB	0x00001000	The TPM permanent flag to allow reading of the Storage Root Key public value is not set.
INFORMATION_TPM_BOOT_COUNTER	0x00002000	The monotonic counter incremented during boot has not been created.
INFORMATION_TPM_AD_BACKUP	0x00004000	The TPM's owner authorization has not been backed up to Active Directory.
INFORMATION_TPM_AD_BACKUP_PHASE_I	0x00008000	The first portion of the TPM owner authorization information storage in Active Directory is in progress.
INFORMATION_TPM_AD_BACKUP_PHASE_II	0x00010000	The second portion of the TPM owner authorization information storage in Active Directory is in progress.

INFORMATION_LEGACY_CONFIGURATION	0x00020000	Windows Group Policy is configured to not store any TPM owner authorization so the TPM cannot be fully ready.
TBD	0x00040000	The EK certificate was not read from the TPM NV RAM and stored in the registry. (See the NCrypt Property NCRYPT_PCP_EKCERT_PROPERTY for more information.)
Symbol	Value	Description
INFORMATION_TCG_EVENT_LOG	0x00080000	The TCG event log is empty or cannot be read. (This is usually a problem with the system firmware, not TPM state.)
INFORMATION_NOT_REDUCED	0x00100000	The TPM is not owned or otherwise not ready for use by BitLocker.
INFORMATION_GENERIC_ERROR	0x00200000	A generic error occurred.

Return value:
If the function succeeds, the function returns S_OK (0).

Remarks:
This method is expensive to run because it performs many checks. It is recommended applications use this method only when necessary.

Provisioning Differences Between TPM Versions 1.2 and 2.0

In terms of provisioning, TPM 1.2 and TPM 2.0 have a variety of differences regarding TPM state and TPM management authorization values. Windows 8 has roughly equivalent actions it takes for both versions of TPM to provision either. Both are auto-provisioned, both use the TPM Management Console and both are able to be manipulated through the WMI Win32_TPM class.

Windows 8 Certified Hardware Requirements

Microsoft has a Windows 8 hardware certification program that helps OEMs ship systems with the best defaults and capabilities for a great customer experience. Windows 8 systems with a TPM should have the following characteristics:

- Implement the TCG Physical Presence Interface Specification 1.2, setting the NoPPIProvision flag to TRUE by default. When ownership of the TPM has not been taken, this flag's setting allows the OS to ready the TPM for use without a physically present user being involved in the provisioning process.
- The TPM will be enumerated to Windows by default. This means Windows sees the Advanced Configuration and Power Interface (ACPI) device object for the TPM and can manage it.
- A full Endorsement Key Certificate provisioned in the TPM's NV RAM using the TPM_NV_INDEX_EKCert index location described in section 4.2.1 of the TCG PC Client Specific Implementation Specification for Conventional BIOS using the structures described in section 7.4 if the TPM does not support generating a new Endorsement Key.

Platform Crypto Provider in Windows 8

Support for the TPM in Windows 8 has been significantly expanded. Crypto-operations is one such area of expanded support. The TPM can now be used for crypto-operations through the standard Crypto Next Generation (CNG) Windows interfaces.

The new CNG-Platform Crypto Provider can use a TPM 1.2 or 2.0 device to provide TPM version-independent crypto services. The provider may be used as a replacement for the Microsoft software provider because it provides a superset of its properties (with the exception of exportable keys that are supported only in an authorized form).

The benefit of using the provider instead of issuing TPM commands directly through the *TPM Based Services* (TBS) is that the provider abstracts TPM version differences and takes care of TPM key management for the application. The provider allows an application to switch between TPM or software based keys by changing to a different provider name. This allows an application provider to keep TPM-specific code to a minimum, meaning the application will also run on machines with no TPM or in cases where the user prefers to not use the TPM.

The Platform Crypto Provider should not be set as default RSA provider on the system because it does not operate with the same performance that the Microsoft software provider does. Also, the software provider supports unauthorized exportable keys that do not make sense for the Platform Crypto Provider.

Certificate Enrollment with the Platform Crypto Provider

To demonstrate certificate enrollment with the TPM-based Platform Crypto Provider, a self-signed certificate can be created with the certreq.exe utility as follows: '`certreq.exe -new PCPCert.inf PCPCert.cer`' where the file PCPCert.inf has the contents:

```
[NewRequest]
   Subject = "CN=SelfSignedPCPCert"
   HashAlgorithm = sha256
   KeyAlgorithm = RSA
   KeyLength = 2048
   KeyUsage = "CERT_DIGITAL_SIGNATURE_KEY_USAGE"
   KeyUsageProperty = "NCRYPT_ALLOW_SIGNING_FLAG"
   ProviderName = "Microsoft Platform Crypto Provider"
   RequestType = Cert
   FriendlyName = "DeleteMe!"
   Exportable = false
[EnhancedKeyUsageExtension]
   OID=2.5.29.37.0
```

In order to use the provider in an enterprise scenario with a Windows 8 Certificate Authority (CA) and Policy Server, the server has to have a TPM that is *ready* and may be using the TPM to protect the CA signing keys, for example. The CA administrator then creates a certificate template that mandates the usage of the Platform Crypto Provider for key storage on the client. Clients that enroll for this certificate will make use of the TPM without any action from their side.

Tracing Provider TPM Commands

Advanced users who have applications that are issuing their own TPM commands or are debugging other complex issues can audit the operations that the provider executes on the TPM and can turn on provider tracing to receive a decoded trace of the TPM communication by setting a REG_SZ value `ProviderTraces` under the key

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM
```

with a folder name. If set, the provider will write TPM trace log files in this directory. The trace log files are identified by the process ID. (The application using the Platform Crypto Provider must have write access to the directory because the provider executes in-process).

Trace examples of a key creation:

TPM12Trace.txt TPM20Trace.txt

Administrator privileges are required to set the registry value because the trace files may contain secrets that are passed to and from the TPM. Note also that this feature will produce substantial amounts of data on the disk.

BCrypt RNG Platform Crypto Provider

The BCrypt Platform Crypto Provider exposes the TPM random number generator (RNG) directly through the CNG RNG interface. This is in contrast to the default Windows RNG provider, which uses many sources of entropy (the TPM is one source).

Supported Functions

To open the provider, call BCryptOpenAlgorithmProvider with *pszAlgId* = BCRYPT_RNG_ALGORITHM and specify the provider with *pszImplementation* = MS_PLATFORM_CRYPTO_PROVIDER.

A call to BCryptGenRandom will fill the caller-provided buffer with random data. The maximum random number size is limited to 4,096 bytes per call. The provider may make several round trips to the TPM to fill the caller's buffer and the call will block until the request is satisfied.

The provider also supports "stirring" the RNG in the TPM, by providing the flag BCRYPT_RNG_USE_ENTROPY_IN_BUFFER with the call BCryptGenRandom. If this flag is set, the provider will stir the entropy generator in the TPM with the data in the caller's buffer. If the buffer is larger than 256 bytes, only the first 256 bytes will be used.

Supported Properties

BCRYPT_ALGORITHM_NAME	Algorithm name as defined in MSDN	GET
BCRYPT_PCP_PLATFORM_TYPE_PROPERTY	String that identifies the TPM manufacturer and version	GET
BCRYPT_PCP_PROVIDER_VERSION_PROPERTY	Crypto Provider Version	GET

NCrypt RSA Platform Key Storage Provider

The Platform Crypto Provider's Key Storage Provider (KSP) provides a subset of the functionality that is provided by the Microsoft software provider. It supports all functionality that is required by certificate enrollment. The Platform Crypto Provider does not support regular exportable keys, because those are by definition not bound to the platform and there would be no security benefit by using the TPM for these keys. However, the provider does support authenticated exportable keys that use the *migrationAuth* for a controlled export. The *migrationAuth* is a Platform Crypto Provider-specific key property that allows the TPM to enforce key export control of key material.

Beyond the well-defined provider and key properties, the Platform Crypto Provider supports a number of special properties that enable specific TPM-related scenarios.

Supported Functions

- NCryptCreatePersistedKey
- NCryptDecrypt
- NCryptDeleteKey
- NCryptEncrypt
- NCryptEnumAlgorithms
- NCryptEnumKeys
- NCryptExportKey
- NCryptFinalizeKey
- NCryptFreeBuffer
- NCryptFreeObject
- NCryptGetProperty
- NCryptImportKey
- NCryptIsAlgSupported
- NCryptOpenKey
- NCryptOpenStorageProvider
- NCryptSetProperty
- NCryptSignHash
- NCryptVerifySignature

Supported Properties

Properties that are exclusively provided by the PCP provider are marked in gray.

Provider Properties		
BCRYPT_ALGORITHM_NAME	Cryptography Primitive Property Identifiers as defined in MSDN	GET
BCRYPT_PADDING_SCHEMES	Cryptography Primitive Property Identifiers as defined in MSDN	GET
BCRYPT_KEY_LENGTHS	Cryptography Primitive Property Identifiers as defined in MSDN	GET
NCRYPT_USE_CONTEXT_PROPERTY	Key Storage Property Identifiers as defined in MSDN	SET
NCRYPT_PCP_PLATFORM_TYPE_PROPERTY	String that identifies the TPM manufacturer and version	GET
NCRYPT_PCP_PROVIDER_VERSION_PROPERTY	Crypto Provider Version	GET
NCRYPT_PCP_EKPUB_PROPERTY	EK_{PUB} from the TPM as a BCRYPT_RSAKEY_BLOB structure	GET
NCRYPT_PCP_EKCERT_PROPERTY	Read/Write HCERTSTORE handle to the EKCert store in the registry. The caller may persist multiple custom-generated EK certs in the store and may delete the TPM manufacturer-provided one if so desired. The TPM manufacturer-provided EK cert may always be read from the property NCRYPT_PCP_EKNVCERT_PROPERTY.	GET
NCRYPT_PCP_EKNVCERT_PROPERTY	EKCert directly from the TPM NVRAM (if present)	GET
NCRYPT_PCP_SRKPUB_PROPERTY	SRK_{PUB} from the TPM as BCRYPT_RSAKEY_BLOB structure	GET
NCRYPT_PCP_PCRTABLE_PROPERTY	All 24 SHA-1 PCRs in ascending order from PCR[0] to PCR[23] returned as a packed array of binary arrays	GET
NCRYPT_PCP_PLATFORMHANDLE_PROPERTY	The TBS handle that the provider uses to submit commands to the TPM. A provider platform key handle is only valid within this TBS context.	GET
NCRYPT_PCP_ALTERNATE_KEY_STORAGE_LOCATION_PROPERTY	Specify an alternate base key storage path for the provider operation	GET

NCRYPT_PROVIDER_HANDLE_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_NAME_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_UNIQUE_NAME_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_MAX_NAME_LENGTH_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_WINDOW_HANDLE_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET; SET
NCRYPT_USE_CONTEXT_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET; SET
NCRYPT_KEY_USAGE_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET; SET
NCRYPT_EXPORT_POLICY_PROPERTY	Key Storage Property Identifiers as defined in MSDN For SET, only NCRYPT_ALLOW_ARCHIVING_FLAG with NCRYPT_ALLOW_PLAINTEXT_ARCHIVING_FLAG is supported. GET will return NCRYPT_ALLOW_EXPORT_FLAG with NCRYPT_ALLOW_PLAINTEXT_EXPORT_FLAG if a key may be exported with an export password.	GET; SET
NCRYPT_UI_POLICY_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET; SET
NCRYPT_KEY_TYPE_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_LENGTH_PROPERTY	Key Storage Property Identifiers as defined in MSDN Only 2,048- (default) and 1,024-bit keys are supported.	GET; SET

NCRYPT_LENGTHS_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_ALGORITHM_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_ALGORITHM_GROUP_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_CERTIFICATE_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET; SET
NCRYPT_LAST_MODIFIED_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET
NCRYPT_SECURITY_DESCR_PROPERTY	Key Storage Property Identifiers as defined in MSDN	GET; SET
NCRYPT_PIN_PROPERTY, BCRYPT_PCP_PASSWORD_PROPERTY	Key Storage Property Identifiers as defined in MSDN	SET
NCRYPT_PCP_MIGRATIONPASSWORD_PROPERTY	Migration password for a key. Setting this property before it is finalized will make a key exportable.	SET
NCRYPT_PCP_PLATFORM_TYPE_PROPERTY	String that identifies the TPM manufacturer and version	GET
NCRYPT_PCP_PROVIDER_VERSION_PROPERTY	Crypto Provider Version	GET
NCRYPT_PCP_STORAGEPARENT_PROPERTY	Optional handle to a storage key that this key is protected by. Has to be set before the key is finalized and every time the key is loaded.	GET; SET
BCRYPT_KEY_LENGTH, BCRYPT_KEY_STRENGTH	Cryptography Primitive Property Identifiers as defined in MSDN	GET; SET
BCRYPT_BLOCK_LENGTH, BCRYPT_SIGNATURE_LENGTH	Cryptography Primitive Property Identifiers as defined in MSDN	GET

NCRYPT_PCP_PLATFORMHANDLE_PROPERTY	Virtualized Key handle that this key uses in the TPM. This property in conjunction with the provider NCRYPT_PCP_PLATFORMHANDLE_PROPERTY may be used to issue custom commands on a loaded key. The handle is only valid within the TBS context that the provider uses.	GET
NCRYPT_PCP_KEY_USAGE_POLICY	TPM key usage restriction and special TPM keys: NCRYPT_TPM12_PROVIDER = (0x00010000) NCRYPT_PCP_SIGNATURE_KEY = (0x00000001) NCRYPT_PCP_ENCRYPTION_KEY = (0x00000002) NCRYPT_PCP_GENERIC_KEY = (NCRYPT_PCP_SIGNATURE_KEY | NCRYPT_PCP_ENCRYPTION_KEY) NCRYPT_PCP_STORAGE_KEY = (0x00000004) NCRYPT_PCP_IDENTITY_KEY = (0x00000008)	GET; SET
NCRYPT_PCP_PASSWORD_REQUIRED_PROPERTY	BOOLEAN value that can be queried to discover if a key requires authentication prior to use	GET
NCRYPT_PCP_EXPORT_ALLOWED_PROPERTY	BOOLEAN value that can be queried to discover if a key is exportable. This value will also return TRUE if a key was generated outside the TPM to indicate that the key material was at least at some point in existence outside the TPM. For key import on TPM 1.2, if no explicit export password was set, a randomly generated unknown value is set for the export password.	GET
NCRYPT_PCP_USAGEAUTH_PROPERTY	TPM authValue for the key (typically the SHA-1 digest of the usage password)	GET; SET
NCRYPT_PCP_TPM12_IDBINDING	For keys with usage policy NCRYPT_PCP_IDENTITY_KEY only. SET will set the nonce from the CA before the key is finalized and after the key is finalized the ID Binding may be retrieved with GET. Only valid on initial key handle. This property is not persisted. The IDBinding returned by this property consists of a structure containing the public portion of the AIK	GET; SET

	along with other data, and a signature over this structure.	
NCRYPT_PCP_TPM12_IDACTIVATION	For keys with usage policy NCRYPT_PCP_IDENTITY_KEY only. SET will set the challenge from the CA and the EK wrapped secret may be retrieved with GET. Administrative privileges are required.	GET; SET
NCRYPT_PCP_PLATFORM_BINDING_PCRMASK_PROPERTY	Before the key is finalized, SET allows a UINT32 mask value to be provided that binds key usage to the indicated set of PCR values. Bit 0 of the mask corresponds with PCR[0] and bit 23 with PCR[23]. If no PCR table is provided, the current PCR values in the TPM are used. GET will return the mask if a key was bound to PCRs.	GET; SET
NCRYPT_PCP_PLATFORM_BINDING_PCRDIGESTLIST_PROPERTY	Provide all 24 SHA-1 PCRs that are to be used for PCR binding during key creation as a packed binary array, with PCR[0] first	SET
NCRYPT_PCP_PLATFORM_BINDING_PCRDIGEST_PROPERTY	Return the PCR digest a key is bound to. This is the SHA-1 digest of the TPM_PCR_COMPOSITE structure on TPM 1.2 and the SHA-256 digest of the PCRs to be provided to the TPM2_PolicyPCR command on TPM 2.0.	GET
NCRYPT_PCP_CHANGEPASSWORD_PROPERTY	Allows changing a usage password on a fully authorized key	SET
NCRYPT_PCP_KEYATTESTATION_PROPERTY	If at least one AIK is registered, the provider will generate key certification data for every non-exportable key that is created. This property allows access to this data after the key is finalized. The attestation data is stored persistent with the key.	GET

Description of Provider Data Structures

BCRYPT_OPAQUE_KEY_BLOB

The Platform Crypto Provider allows export and import of TPM-protected key blobs for backup and restore purposes or to use that key blob directly in a TPM command. This functionality is only available on the Platform Crypto Provider.

```
#define BCRYPT_PCP_KEY_MAGIC 'MPCP' // Platform Crypto Provider Magic
#define PCPTYPE_TPM12 (0x00000001)  // TPM type 1.2
#define PCPTYPE_TPM20 (0x00000002)  // TPM type 2.0
```

On TPM 1.2 systems, the key blob has a PCP_KEY_BLOB header followed by the indicated data in order:

```
typedef enum PCP_KEY_FLAGS {
    PCP_KEY_FLAGS_authRequired = 0x00000001   // Key uses authorization
} PCP_KEY_FLAGS;

typedef struct PCP_KEY_BLOB
{
    DWORD   magic;              // BCRYPT_PCP_KEY_MAGIC
    DWORD   cbHeader;           // Size of the header structure
    DWORD   pcpType;            // TPM type
    DWORD   flags;              // PCP_KEY_FLAGS Key flags
    ULONG   cbTpmKey;           // Size of TPM_KEY12 structure
} PCP_KEY_BLOB, *PPCP_KEY_BLOB;
```

On TPM 2.0 systems, the key blob has a PCP_KEY_BLOB_WIN8 header followed by the indicated data in order:

```
typedef enum PCP_KEY_FLAGS_WIN8 {
    PCP_KEY_FLAGS_WIN8_authRequired = 0x00000001  // Key uses authorization
} PCP_KEY_FLAGS_WIN8;

typedef struct PCP_KEY_BLOB_WIN8
{
    DWORD   magic;              // BCRYPT_PCP_KEY_MAGIC
    DWORD   cbHeader;           // Size of the header structure
    DWORD   pcpType;            // TPM type
    DWORD   flags;              // PCP_KEY_FLAGS_WIN8 Key flags
    ULONG   cbPublic;           // Size of Public key
    ULONG   cbPrivate;          // Size of Private key blob
    ULONG   cbMigrationPublic;  // Size of Public migration authorization object
    ULONG   cbMigrationPrivate; // Size of Private migration authorization object
    ULONG   cbPolicyDigestList; // Size of List of policy digest branches
    ULONG   cbPCRBinding;       // Size of PCR binding mask
    ULONG   cbPCRDigest;        // Size of PCR binding digest
    ULONG   cbEncryptedSecret;  // Size of hostage import symmetric key
    ULONG   cbTpm12HostageBlob; // Size of hostage import private key
} PCP_KEY_BLOB_WIN8, *PPCP_KEY_BLOB_WIN8;
```

NCRYPT_PCP_TPM12_IDBINDING

SET only accepts an SHA-1 digest on TPM 1.2 and 2.0 systems.

On TPM 1.2 systems, GET provides the following data after key finalize:

TPM_IDENTITY_CONTENTS || SIGNATURE

On TPM 2.0 systems, GET provides the following data after key finalize:

TPM2B_PUBLIC || TPM2B_CREATION_DATA || TPM2B_ATTEST || TPMT_SIGNATURE

Automatic Key Attestation

The attestation data that is returned from the property NCRYPT_PCP_KEYATTESTATION_PROPERTY is formatted as follows. It starts with a tag identifying the version, followed by attestation packages.

```
    ULONG    tag                // 'AK1T' for 1.2 and 'AK2T' for 2.0 TPM data
{
    ULONG    attestationSize    // There will be one attestation section for each
                                   registered Attestation Key
    USHORT   sizeAikName
    WCHAR    keyName[]          // Name under which the key is stored in the registry
    USHORT   sizeAikPubDigest
    BYTE     aikPubDigest[]     // SHA-1 Digest for the AIK modulus
    USHORT   sizeAttestationData
    BYTE     attestationData[]  // TPM-generated attestation structure
    USHORT   sizeSignature
    BYTE     signature[]       // AIK signature over the SHA-1 digest of attestation data
}
{
  // Second attestation package as above
}
```

NCRYPT_PCP_TPM12_IDACTIVATION

Use SET on TPM 1.2 systems to provide the EK_{PUB} encrypted structure TPM_EK_BLOB that contains the structure TPM_EK_BLOB_ACTIVATE.

Use SET on TPM 2.0 systems to set the activation credential:

TPM2B_ID_OBJECT || TPM2B_ENCRYPTED_SECRET

Use GET to perform the activation and retrieve the wrapped secret.

Using Storage Provider Keys with Custom Commands

If an application needs to run TPM commands that are not supported by the crypto provider, the provider can be queried for the relevant handles and contexts.

1. Open the crypto provider and create or open one or more keys to be used in a custom command.
2. Obtain the TBS handle hTbsContext from the provider with NCryptGetProperty(hProvider, NCRYPT_PCP_PLATFORMHANDLE_PROPERTY,…).
3. Obtain each key handle required for the custom command NCryptGetProperty($hKey_n$, NCRYPT_PCP_PLATFORMHANDLE_PROPERTY,…).
4. Get the TPM version on the platform from the TBS with the new function Tbsi_GetDeviceInfo().
5. Format the TPM-specific command (1.2 or 2.0 depending on what the previous call returned) structure using the obtained key handles.
6. Submit the custom command to the TPM with Tbsip_Submit_Command(hTbsContext, …).

Do not close the hTbsContext or the virtualized key handles in the TPM because this will put the provider in an undefined state. Closing the keys through the provider and closing the provider itself will release all resources.

Executing Custom TPM Commands Through the TBS API

TPM Base Services (TBS) is the Windows facility that allows commands to be sent to the underlying TPM without the abstraction provided by BCrypt. This section describes TBS functionality as well as auxiliary services such as access to the Windows boot configuration log (WBCL) and intentionally invalidating attestation.

TBS requires the calling application to prepare properly formatted TPM command buffers that are compatible with the underlying TPM (TPM 1.2 or TPM 2.0). However, Windows provides services that allow the TPM and its internal resources to be shared. These facilities are described in the next section.

TPM Resource Virtualization

One service provided by TBS is sending preformatted TPM commands to the TPM and providing the response back to the application. To isolate multiple applications from each other with respect to their effects on TPM state, TBS implements TPM context management. The TPM context management virtualizes certain limited TPM resources, and restricts access to these resources to the applications that created them.

To support this, TBS provides the notion of a *TPM context*, a resource container holding transient TPM resources that have been created by an application. Applications must first create such a TPM context, or use a previously created context such as that provided by the BCrypt provider. To run commands on the TPM, applications will then supply the TPM context together with the preformatted TPM command buffers.

TPM commands and responses contain *handles* for resources that are used or created. When an application sends a command to TBS to be run on the TPM, TBS examines the command. TBS ensures that transient resources referenced by the command have been created within the associated TPM context. On the way back, the TBS analyzes the TPM response to learn about newly created resources. If the limited resource slots of the TPM are filled, TBS will also transparently context-save some resources out of the TPM to make room for executing the current command. When these context-saved resources are referenced later by another TPM command, TBS will transparently reload these resources before executing the command (potentially context-saving other resources). This transparent context-saving and context-loading of resources is referred to as *resource management*.

Because TPM resource handles for transient objects change whenever these objects are reloaded into the TPM, TBS provides its own per-context handle namespace. Whenever a transient resource is created by a TPM command, TBS assigns a handle that uniquely identifies this resource within the caller's context. TBS replaces the handle in the original TPM response, and passes the modified response back to the application. Similarly, when an application sends a TPM command to TBS, TBS will replace the handles with the currently valid TPM handles for the referenced resources (after potentially context-loading these resources). This resource management, combined with the resource handle replacement, is referred to as *resource virtualization*.

The following table lists the type of resources that the TBS manages and the corresponding actions:

Resource Type	Handle Type	TBS Action
TPM1.2 Resources		Resource virtualization
TPM2.0 HMAC Session	0x02	Resource management
TPM2.0 Policy Session	0x03	Resource management
TPM2.0 Transient Object	0x80	Resource virtualization

Command Filtering for 1.2 and 2.0

When inspecting the TPM commands sent by applications, TBS will also apply filtering based on the TPM command code of the command. The TBS will block a command from running according to the following criteria:

- TPM 1.2: If the command code is on one of the blocked lists, the command will be blocked. Note that these lists can be modified using the appropriate WMI interfaces or using the Windows TPM management console.
- TPM 2.0: If the command code is not on one of the allow lists, and the command code does not specify a vendor-specific command, the command will be blocked. Vendor-specific commands will be allowed.

Note that the above blocked and allow lists differentiate between standard users and users with administrative access, and can be modified by Group Policy. In the standard Windows configuration, all TPM commands required for attestation are allowed.

TBS API

TBS provides the following function calls that are relevant for attestation. The functions are described in detail in the following sections.

```
Tbsi_Context_Create
Tbsip_Context_Close
Tbsi_Get_TCG_Log
Tbsi_Revoke_Attestation
Tbsi_GetDeviceInfo
Tbsip_Submit_Command
```

Creating TPM 1.2 and 2.0 Contexts

Most TBS functions require a TBS context, an object used by Windows to manage virtualized TPM resources. The TBS context serves as a handle to these TBS functions.

An application creates a TBS context using the **Tbsi_Context_Create()** function:

```
TBS_RESULT WINAPI Tbsi_Context_Create(
    __in  PCTBS_CONTEXT_PARAMS pContextParams,
    __out PTBS_HCONTEXT        phContext);
```

Parameters:

`pContextParams [in]` – A parameter to a **TBS_CONTEXT_PARAMS** structure that contains the parameters associated with the context. See the Remarks section below.

`phContext [out]` – A pointer to a location to store the new context handle.

Return value:

If the function succeeds, the function returns TBS_SUCCESS (0).

Remarks:

The *pContextParams* parameter allows the caller to specify the TPM version (TPM 1.2 or TPM 2.0) that it is prepared to interact with. For applications interacting with TPM version 1.2 only, a pointer to a **TBS_CONTEXT_PARAMS** can be provided, with the *version* field set to **TPM_VERSION_12**.

Applications interacting with TPM version 2.0 will pass a pointer to a **TBS_CONTEXT_PARAMS2** structure, with the *version* field set to **TPM_VERSION_20**. Set the *reserved* field to 0, and the *includeTPm20* field to 1. If the application is prepared to interact with TPM version 1.2 as well (in case the system has no TPM version 2.0), set the *includeTpm12* field to 1.

```
#define TPM_VERSION_12          1
#define TPM_VERSION_20          2

typedef struct {
    UINT32 version;
} TBS_CONTEXT_PARAMS, *PTBS_CONTEXT_PARAMS;
typedef const TBS_CONTEXT_PARAMS *PCTBS_CONTEXT_PARAMS;

typedef struct {
    UINT32  version;
    UINT32  reserved : 1;
    UINT32  includeTpm12 : 1;
    UINT32  includeTpm20 : 1;
} TBS_CONTEXT_PARAMS2, *PTBS_CONTEXT_PARAMS2;
typedef const TBS_CONTEXT_PARAMS2 *PCTBS_CONTEXT_PARAMS2;
```

If no TPM is present on the system, or the TPM version does not match those requested by the caller, **Tbsi_Context_Create()** will return the **TBS_E_TPM_NOT_FOUND** (0x8028400f) error code. Application programs must check the TPM version and be able to interact with either.

Deleting TPM 1.2 and 2.0 Contexts

An application can free a TBS context and release the associated system resources using the **Tbsip_Context_Close()** function:

```
TBS_RESULT WINAPI Tbsip_Context_Close(
    __in TBS_HCONTEXT   hContext);
```

Parameters:

`hContext [in]` – TBS context handle to free.

Return value:

If the function succeeds, the function returns TBS_SUCCESS (0).

Obtaining the Windows Boot Configuration Log (WBCL)

An application can obtain the WBCL using the Tbsi_Get_TCG_Log() function:

```
TBS_RESULT WINAPI Tbsi_Get_TCG_Log(
    __in    TBS_HCONTEXT hContext,
    __out_bcount_part_opt(*pcbOutput, *pcbOutput)
            PBYTE        pabOutput,
    __inout PUINT32      pcbOutput);
```

Parameters:

`hContext [in]` – TBS handle obtained from a previous call to **Tbsi_Context_Create()**.

`pabOutput [out]` – A pointer to a location to store the WBCL. This parameter may be NULL to estimate the required buffer when the location pointed to by pcbOutput is also 0 on input.

`pcbOutput [in,out]` – A pointer to a location that, on input, specifies the size, in bytes, of the output buffer. If the function succeeds, this parameter, on output, receives the size, in bytes, of the data pointed to by pabOutput. Calling the **Tbsi_Get_TCG_Log()** function with a zero length buffer will return the size of the buffer required.

Return value:

If the function succeeds, the function returns TBS_SUCCESS (0). If the *Size* parameter is too small, the function returns TBS_E_INSUFFICIENT_BUFFER (0x80284005).

Invalidating the System Trust State

If the ELAM driver detects a policy violation (a rootkit, for example), it can invalidate the PCRs that indicated that the system was in a good state using the **Tbsi_Revoke_Attestation()** function:

```
TBS_RESULT WINAPI
Tbsi_Revoke_Attestation();
```

Parameters:

None.

Return value:

If the function succeeds, the function returns TBS_SUCCESS (0).

Remarks:

This function, executable by users with administrative rights, extends PCR[12] by an unspecified value and increments the event counter in the TPM. Both actions are necessary, so the trust is broken in all quotes that are created from here on forward. Because the PCRs are reset on hibernation and the extend to PCR[12] then will disappear, a gap in the event counter will indicate a broken chain of logs.

As a result, the WBCL files will not reflect the current state of the TPM for the remainder of the time that the TPM is powered up and remote systems will not be able to form trust in the security state of the system. Note that anti-malware systems will probably perform additional remediation or alerts, but the invalidation step is crucial if attestation is supported.

When the machine goes to hibernation and subsequently resumes, the above PCR extend will be lost, and the broken trust will not be reflected in the PCR measurements anymore. To address this, **Tbsi_Revoke_Attestation()** also increments the monotonic Event Counter located in the TPM. Further TPM attestation validations will notice a gap in the archived WBCL logs' boot counter values. Upon discovery of such a gap, attestation validation code should fail the validation, just as it would if other required events were not present in the log. Note that the counter in the TPM cannot be rolled back, and hence the missing WBCL cannot be constructed after the fact. The log is described in more detail in the section "Windows Boot Configuration Log."

Obtaining the TPM Version

An application can obtain the version of the TPM on the system using the **Tbsi_GetDeviceInfo()** function:

```
TBS_RESULT WINAPI Tbsi_GetDeviceInfo(
    __in                UINT32           Size,
    __out_bcount(Size) PVOID            Info);
```

Parameters:

Size [in] – Size of the Info memory location.

Info [out] – A pointer to a location to store the version information about the TPM. The location must be large enough to hold four 32-bit values. For details, see the Remarks section below.

Return value:

If the function succeeds, the function returns TBS_SUCCESS (0). If no TPM is present on the system, the function returns TBS_E_TPM_NOT_FOUND (0x8028400f). If the *Size* parameter is too small, the function returns TBS_E_BAD_PARAMETER (0x80284002).

Remarks:

Upon success, the location pointed to by Info will be set according to the following table:

Offset	Size	Comment
0	4	Version of the Info structure. Will be set to 1.
4	4	TPM version. Will be set to TPM_VERSION_12 or TPM_VERSION_20.
8	4	Reserved.
12	4	Reserved.

Submitting a Custom TPM Command

An application can submit a preformatted command to TBS to be run on the TPM using the **Tbsip_Submit_Command()** function:

```
TBS_RESULT WINAPI
Tbsip_Submit_Command(
    __in                        TBS_HCONTEXT         hContext,
    __in                        TBS_COMMAND_LOCALITY Locality,
    __in                        TBS_COMMAND_PRIORITY Priority,
    __in_bcount(cbCommand)      PCBYTE               pabCommand,
    __in                        UINT32               cbCommand,
    __out_bcount(*pcbResult)    PBYTE                pabResult,
    __inout                     PUINT32              pcbResult);
```

Parameters:

hContext [in] – TBS handle obtained from a previous call to **Tbsi_Context_Create()**.

Locality [in] – Must be TBS_COMMAND_LOCALITY_ZERO (0).

Priority [in] – The priority level that the command should have.

pabCommand [in] – A pointer to a buffer that contains the TPM command to process.

cbCommand [in] – The length, in bytes, of the command.

pabResult [out] – A pointer to a buffer to receive the result of the TPM command.

pResultBufLen [in, out] – A pointer to an integer that, on input, specifies the size, in bytes, of the result buffer. On successful return, this value is set to the actual size of the TPM response, in bytes.

Return value:

If the function succeeds, the function returns TBS_SUCCESS (0). Other values indicate errors detected by the TBS during the running of the command. This function can succeed (indicating that the command was sent to the TPM and the TPM responded), but the TPM could have returned an error. In this case, this function returns TBS_SUCCESS, and the TPM failure code is returned as a standard TPM response code in the result buffer.

EXHIBIT N

About TBS

05/31/2018 • 2 minutes to read •

The TPM Base Services (TBS) feature is a system service that allows transparent sharing of the Trusted Platform Module (TPM) resources. It simultaneously shares the TPM resources among multiple applications on the same physical machine, even if those applications run on different virtual machines. Starting with Windows 8 and Windows Server 2012, TBS comes pre-installed on all systems with a TPM.

The Trusted Computing Group (TCG) defines a Trusted Platform Module that provides cryptographic functions designed to provide trust in the platform. Because the TPM is implemented in hardware, it has finite resources. The TCG also defines a software stack that makes use of these resources to provide trusted operations for application software. However, no provision is made for running a TSS implementation side-by-side with operating system software that may also be using TPM resources. The TBS feature solves this problem by enabling each software stack that communicates with TBS to use TPM resources checking for any other software stacks that may be running on the machine.

The TPM specification and TCG Software Stack (TSS) specification are available at https://www.trustedcomputinggroup.org .

TBS is implemented as an out-of-process service that accepts commands that use an RPC service. A dynamically linked library presents the C language interface and communicates with the TBS service.

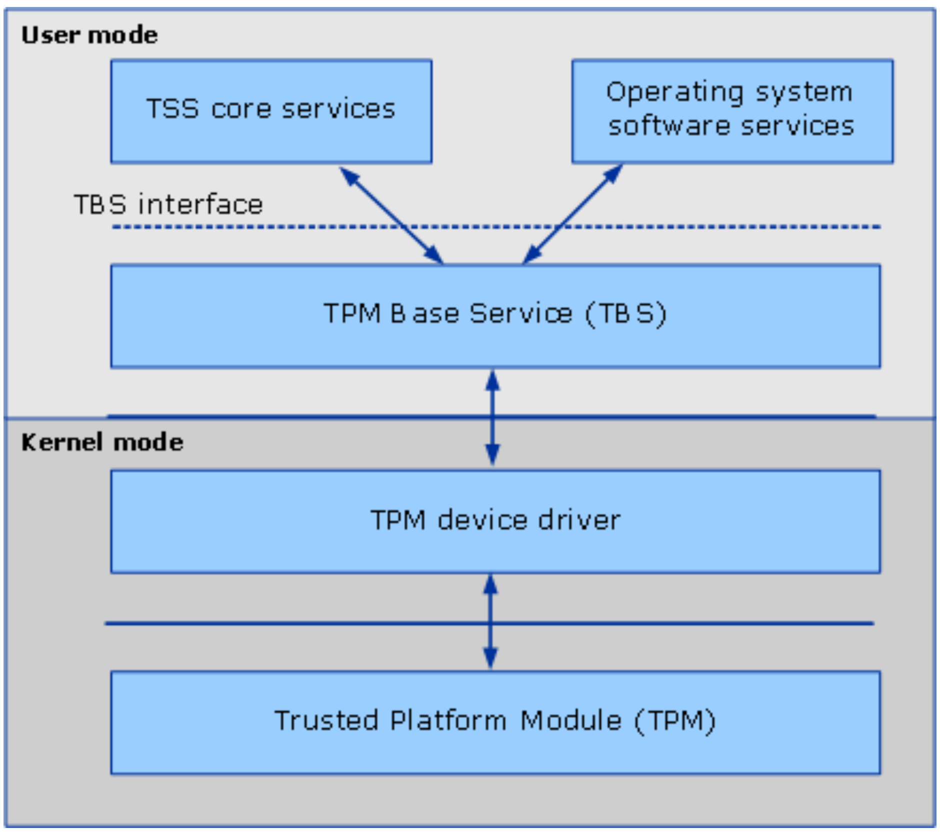
Note

The TBS service only accepts RPC requests from the local machine.

The primary goals of the TBS are to:

- Provide efficient sharing of limited TPM resources, such as key slots, authorization sessions slots, and transport slots.
- Provide prioritized and synchronized access to TPM resources between multiple instances of TPM software stacks.
- Provide appropriate management of TPM resources across power states.
- Prevent TPM software stacks from accessing TPM commands that should be restricted, either because of platform limitations or administrative requirements.

The following illustration shows the relationship of the TBS to the TPM.



Is this page helpful?

Yes No

EXHIBIT O

Tbsip_Submit_Command function (tbs.h)

12/05/2018 • 2 minutes to read

In this article

Submits a Trusted Platform Module (TPM) command to TPM Base Services (TBS) for processing.

Syntax

C++ Copy

```
TBS_RESULT Tbsip_Submit_Command(
  TBS_HCONTEXT         hContext,
  TBS_COMMAND_LOCALITY Locality,
  TBS_COMMAND_PRIORITY Priority,
  PCBYTE               pabCommand,
  UINT32               cbCommand,
  PBYTE                pabResult,
  PUINT32              pcbResult
);
```

Parameters

`hContext`

The handle of the context that is submitting the command.

`Locality`

Used to set the locality for the TPM command. This must be one of the following values.

Value	Meaning
TBS_COMMAND_LOCALITY_ZERO 0 (0x0)	Locality zero. This is the only locality currently supported.
TBS_COMMAND_LOCALITY_ONE 1 (0x1)	Locality one.
TBS_COMMAND_LOCALITY_TWO 2 (0x2)	Locality two.
TBS_COMMAND_LOCALITY_THREE 3 (0x3)	Locality three.
TBS_COMMAND_LOCALITY_FOUR 4 (0x4)	Locality four.

`Priority`

The priority level that the command should have. This parameter can be one of the following values.

Value	Meaning
TBS_COMMAND_PRIORITY_LOW 100 (0x64)	Used for low priority application use.
TBS_COMMAND_PRIORITY_NORMAL 200 (0xC8)	Used for normal priority application use.
TBS_COMMAND_PRIORITY_SYSTEM 400 (0x190)	Used for system tasks that access the TPM.

TBS_COMMAND_PRIORITY_HIGH 300 (0x12C)	Used for high priority application use.
TBS_COMMAND_PRIORITY_MAX 2147483648 (0x80000000)	Used for tasks that originate from the power management system.

`pabCommand`

A pointer to a buffer that contains the TPM command to process.

`cbCommand`

The length, in bytes, of the command.

`pabResult`

A pointer to a buffer to receive the result of the TPM command. This buffer can be the same as *pabCommand*.

`pcbResult`

An integer that, on input, specifies the size, in bytes, of the result buffer. This value is set when the submit command returns. If the supplied buffer is too small, this parameter, on output, is set to the required size, in bytes, for the result.

Return value

If the function succeeds, the function returns TBS_SUCCESS.

A command can be submitted successfully and still fail at the TPM. In this case, the failure code is returned as a standard TPM error in the result buffer.

If the function fails, it returns a TBS return code that indicates the error.

Return code/value	Description
TBS_SUCCESS 0 (0x0)	The function was successful.

TBS_E_BAD_PARAMETER 2150121474 (0x80284002)	One or more parameter values are not valid.
TBS_E_BUFFER_TOO_LARGE 2150121486 (0x8028400E)	The input or output buffer is too large.
TBS_E_INTERNAL_ERROR 2150121473 (0x80284001)	An internal software error occurred.
TBS_E_INSUFFICIENT_BUFFER 2150121477 (0x80284005)	The specified output buffer is too small.
TBS_E_INVALID_CONTEXT 2150121476 (0x80284004)	The specified context handle does not refer to a valid context.
TBS_E_INVALID_OUTPUT_POINTER 2150121475 (0x80284003)	A specified output pointer is not valid.
TBS_E_IOERROR 2150121478 (0x80284006)	An error occurred while communicating with the TPM.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	tbs.h
Library	Tbs.lib

DLL	Tbs.dll

Is this page helpful?

Yes No